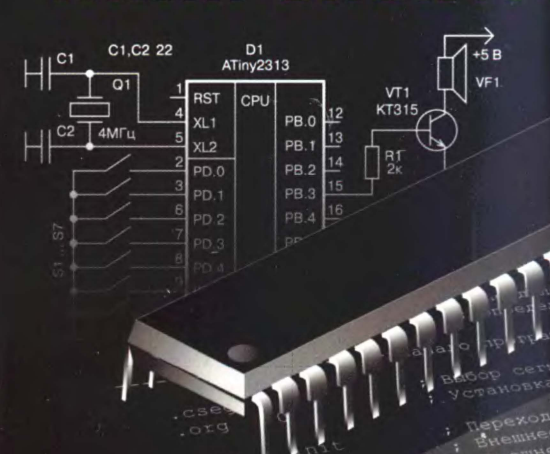
А.В. Белов

Создаем устройства на МИКРОКОНТРОЛЛЕРАХ



Популярная серия микроконтроллеров AVR фирмы Atmel

Учимся создавать свои собственные микропроцессорные устройства и писать для них программы. Просто о сложном.



Бонус: таблица команд Ассемблера микроконтроллеров AVR

Лучшие книги по микроконтроллерам



Самоучитель по микропроцессорной технике

издание 2-е, переработанное и дополненное

Это переработанное издание популярного учебника по микроэлектронике: от принципов работы простейшей электронной логики до структуры микропроцессорного устройства и принципов его работы.

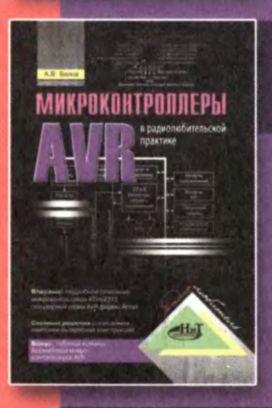
Самоучитель написан доступно, снабжен схемами, иллюстрациями и примерами.



Создаем устройства на микроконтроллерах

Отличное практическое пособие по разработке схем с применением новейших микроконтроллеров.

Основа книги – практические примеры, которые от простого к сложному раскрывают хитрости написания программ, рассматривают их трансляцию и отладку, использование программатора.



Микроконтроллеры AVR в радиолюбительской практике

Данная книга представляет собой справочник, где детально рассмотрен микроконтроллер ATiny2313 семейства AVR.

Имеется практический раздел для радиолюбителей: схемы устройств, выполненных на его базе с описанием и примером управляющей программы на Ассемблере и СИ.



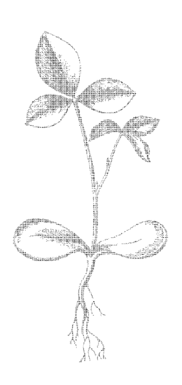
www. nit.com.ru

Россия: С.-Петербург, пр. Обуховской обороны, 107,
для писем: 192029 Санкт-Петербург а/я 44
(812)-567-70-25, 567-70-26, E-mail: nit@mail.wplus.net

Украина: 02166, Киев-166, ул. Курчатова, 9/21,
(044)-516-38-66, E-mail: nits@voliacable.com

А. В. Белов

Создаем устройства на микроконтроллерах





Наука и Техника
Санкт-Петербург
2007

Белов А. В.
Создаем устройства на микроконтроллерах. — СПб.: Наука и Техника, 2007. — 304 с.: ил.

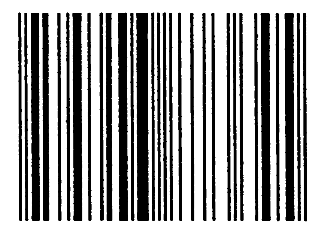
ISBN 978-5-94387-364-3

Серия «Радиолюбитель»

Данная книга представляет собой практическое пособие по разработке электронных схем с применением микроконтроллеров и управляющих программ к ним. Основа книги – это ряд практических примеров, которые от простого к сложному раскрывают принципы построения схем и написания программ для микроконтроллеров. Специально разработанные примеры вводят читателя в мир программирования с самых азов, пройти по всем этапам усложнения задачи и заканчиваются описанием нескольких интересных конструкций имеющих определенную практическую ценность.

После урока по программированию и схемотехники читатель получает подробные сведения о том, как происходит написание трансляция и отладка программ, познакомится с программными средствами, облегчающими редактирование и отладку программ. В заключении вы познакомитесь с принципами построения программаторов для прошивки оттранслированных программ в микросхему микроконтроллера, рассмотрите конкретную схему программатора и научитесь работать с программой, управляющей этим программатором.

Книга рассчитана на широкий круг читателей. Она будет полезна разработчикам электронных устройств, радиолюбителям и студентам технических ВУЗов.



ISBN 978-5-94387-364-3

Контактные телефоны издательства
(812) 567-70-25, 567-70-26
(044) 516-38-66

Официальный сайт: www.nit.com.ru

ООО «Наука и Техника».
Лицензия №000350 от 23 декабря 1999 года.
198097, г. Санкт-Петербург, ул. Маршала Говорова, д. 29.

Подписано в печать 01.02.2007. Формат 60×88 1/16.
Бумага газетная. Печать офсетная. Объем 19 п. л.
Тираж 5000 экз. Заказ № 911.

Отпечатано с готовых диапозитивов в ОАО «Техническая книга»
190005, Санкт-Петербург, Измайловский пр., 29

Содержание

Для заметок

Эту книгу посвящаю моей маме,
Беловой Вере Павловне

Вступление

Современную микроэлектронику трудно представить без такой важной составляющей, как микроконтроллеры. Микроконтроллеры незаметно завоевал весь мир. В последнее время на помощь человеку пришла целая армия электронных помощников. Мы привыкли к ним и часто даже не подозреваем, что во многих таких устройствах работает микроконтроллер.

Микроконтроллерные технологии очень эффективны. Одно и то же устройство, которое раньше собиралось на традиционных элементах, будучи собрано с применением микроконтроллеров, становится проще. Оно не требует регулировки и меньше по размерам.

Кроме того, с применением микроконтроллеров появляются практически безграничные возможности по добавлению новых потребительских функций и возможностей к уже существующим устройствам. Достаточно просто поменять программу!

Где же применяются микроконтроллеры? Да просто везде! Посмотрите вокруг себя. У вас в квартире стоит современный телевизор? Не сомневайтесь: в нем есть, как минимум, один микроконтроллер. У вас есть на руке электронные часы? Современные часы — это просто специализированный микроконтроллер.

Ну, а мобильные телефоны — это, вообще, миниатюрные компьютеры! Возможно, у вас есть игровая приставка, карманная электронная игра, современная микроволновая печь, стиральная машина, проигрыватель лазерных дисков, калькулятор. Во всех этих устройствах работает микроконтроллер. Микроконтроллер применяется и в бытовых приборах, и в сложных промышленных установках.

Однако задача разработки радиоэлектронных устройств с применением микроконтроллеров требует знания и понимания принципов их работы, но главное — умение составлять управляющие программы. Без программы микроконтроллер просто кусочек пластмассы с ножками. В данной книге мы научимся разрабатывать микроконтроллерные устройства.

Мы будем ставить перед собой задачи, начиная от самых простых, имеющих ценность только лишь как элементарные примеры. Постепенно будем задачи усложнять, дойдем до вполне реальных практических разработок.

В качестве базовой микросхемы для всех приведенных в книге примеров использована микросхема ATiny2313 популярной микропроцессорной серии AVR фирмы Atmel. Предполагается, что читатель имеет основные представления, как о принципах построения микропроцессорной техники, так и об основных особенностях архитектуры микроконтроллеров семейства AVR.

Если вы не обладаете подобными знаниями, рекомендую обратиться к новому изданию Самоучителя по микропроцессорной технике [3], где все это описывается достаточно подробно. Полное и подробное описание микросхемы ATiny2313 вы можете найти в моей новой книге Микроконтроллеры AVR в радиолюбительской практике [4].

Рассмотрим теперь книгу, лежащую перед вами. Первая глава представляет собой ряд примеров, при помощи которых шаг за шагом раскрываются основные секреты программирования. Вы начнете с простейшей программы и закончите такими устройствами, как музыкальная шкатулка и кодовый дверной замок. Специально разработанные примеры построены таким образом, что представляют собой ряд уроков программирования, позволяющих изучить язык программирования от практически нулевого уровня, до уровня, позволяющего писать программы средней сложности.

Каждый новый пример начинается с постановки задачи. Затем вы можете увидеть процесс построения алгоритма. Далее мы вместе создадим электрическую схему, и, наконец, увидим, как создается управляющая программа для этой схемы.

Все программные примеры приведены в двух вариантах: на языке Ассемблера и на языке СИ. Каждый пример снабжен подробным описанием. И это не просто сухое описание программ.

Вы получите ряд уроков, из которых узнаете:

- основные приемы программирования на Ассемблере и языке СИ;
- правила построения программ.

Вы сможете сравнить два этих языка, оценить основные преимущества и недостатки каждого из них.

Во второй главе книги мы научимся работать в программной среде AVR Studio и среде Code Vision. Если выражаться простым языком,

то каждая из указанных выше программных сред представляет собой компьютерную программу, специально предназначенную для написания и отладки программ для микроконтроллеров фирмы AVR.

Программа AVR Studio разработана фирмой Atmel и позволяет создавать, транслировать и отлаживать программы на Ассемблере. Программа Code Vision позволяет создавать и отлаживать программы на языке СИ.

В конце второй главы книги вы познакомитесь с методами прошивки программ в программную память микросхемы микроконтроллера. Будет подробно описан популярный программатор PonyProg, его схема и управляющая программа.

Автор надеется, что настоящая книга будут полезна широкому кругу начинающих конструкторов электронной техники, радиолюбителей и студентов технических вузов, и будет благодарен за любые замечания и комментарии по книге. Все замечания прошу высылать по адресу Украина, г. Симферополь, ул. Русская, 194 или по E-mail: avbelov@by.ru.

Дополнительную информацию об этой, а также о других моих книгах вы можете почерпнуть на специальном сайте поддержки моих книг по адресу: http://book.microprocessor.by.ru/

Ознакомиться с новинками и приобрести книги из любой страны мира можно через официальный сайт и Интернет-магазин издательства Наука и Техника www.nit.com.ru.

Удачи Вам во всех делах!

ГЛАВА 1

Написание программ для микроконтроллеров AVR

В этой главе мы научимся создавать свои собственные микропроцессорные устройства и писать для них программы. Данная глава представляет собой практический курс по конструированию простейших устройств на микроконтроллерах.

Прочитав эту главу вы научитесь выполнять постановку задачи, составлять алгоритм работы микропроцессоного устройства, разрабатывать электрическую схему для конкретной задачи, создавать программу на языке Ассемблера, создавать программу на языке СИ.

1.1. Общие положения

Главная задача этой книги — научиться создавать программы для микроконтроллеров. Как можно узнать из [3] программа для микроконтроллера — это набор кодов, который записывается в его специальную программную память. Программу должен написать программист, который разрабатывает ту или иную конкретную микропроцессорную систему.

Однако программист никогда не имеет дело с кодами. Часто программист даже и не задумывается о том, какой код соответствует той или иной команде. Дело в том, что для человека программирование в кодах очень неудобно. Человек же не компьютер.

Для человека удобнее оперировать с командами, каждой из которых имеет свое осмысленное название. Поэтому для написания программ человек использует языки программирования.

Определение. ***Язык программирования** — это специально разработанный язык, служащий посредником между машиной и человеком. Как и обычный человеческий язык, любой язык программирования имеет свой словарь (набор слов) и правила их написания.*

В качестве слов в языке программирования выступают:

- команды (операторы);
- специальные управляющие слова;
- названия регистров;
- числовые выражения.

Главная задача языка — однозначно описать последовательность действий, которую должен выполнить ваш микроконтроллер. В то же время язык должен быть удобен и понятен человеку.

В процессе создания программы программист просто пишет ее текст на компьютере точно так же, как он пишет любой другой текст. Затем программист запускает специальную программу — транслятор.

Определение. ***Транслятор** — это специальная программа, которая переводит текст, написанный программистом, в машинные коды, то есть в форму, понятную для микроконтроллера.*

Написанный программистом текст программы называется **исходным** или **объектным кодом**. Код, полученный в результате трансляции, называется **результирующим** или **машинным кодом**. Именно этот код записывается в программную память микроконтроллера. Для записи результирующего кода в программную память применяются специальные устройства — **программаторы**. О программаторах мы подробно поговорим в последней главе этой книги.

Все языки программирования делятся на две группы:

- языки низкого уровня (машиноориентированные);
- языки высокого уровня.

Типичным примером машиноориентированного языка программирования является **язык Ассемблер**. Этот язык максимально приближен к системе команд микроконтроллера. Каждый оператор этого языка — это, по сути, словесное название какой-либо конкретной команды.

В процессе трансляции такая команда просто **заменяется кодом операции**. Составляя программу на языке Ассемблер, программист должен оперировать теми же видами данных, что и сам процессор, то есть байтами и битами.

Специфика языка Ассемблер состоит еще и в том, что набор операторов для этого языка напрямую зависит от системы команд конкретного микроконтроллера. Поэтому, если два микроконтроллера имеют разную систему команд, то и язык Ассемблер для каждого такого микроконтроллера будет свой. В данной книге мы будем изучать одну конкретную версию языка Ассемблер. А именно **Ассемблер для микроконтроллеров AVR**.

В недавнем прошлом язык Ассемблер был единственным языком программирования для микроконтроллеров. Только он позволял эффективно использовать скудные ресурсы самых первых микросхем. Однако в настоящее время, когда возможности современных микроконтроллеров значительно возросли, для составления программ все чаще используются языки высокого уровня, такие как **Бейсик**, **СИ** и т. п.

Эти языки в свое время были разработаны для больших настоящих компьютеров. Но сейчас широко используются также и для микроконтроллеров. Языки высокого уровня отличаются тем, что они гораздо больше ориентированы на человека. Большинство команд языков высокого уровня не связаны с конкретными командами микроконтроллера.

Такие языки оперируют уже не с байтами, а с привычными нам десятичными числами, а также с переменными, константами и другими элементами, знакомыми нам из математики. Константы и переменные могут принимать привычные для нас значения.

Например, положительные, отрицательные значения, вещественные значения (десятичные дроби) и т. п. Со всеми переменными и константами можно выполнять знакомые нам арифметические операции и даже алгебраические функции.

Транслятор с языка высокого уровня производит более сложные преобразования, чем транслятор с Ассемблера. Но в результате тоже получается **программа в машинных кодах**. При этом транслятор использует все ресурсы микроконтроллера по своему усмотрению. В каких именно регистрах или ячейках памяти она будет хранить значения описанных вами переменных, по каким алгоритмам она будет вычислять математические функции, программист обычно не задумывается.

Программа-транслятор выбирает все это сама. Поэтому задача эффективности алгоритма полученной в результате трансляции программы целиком ложится на программу-транслятор. В целом, программы, написанные на языках высокого уровня, занимают в памяти микроконтроллера объем на 30—40 % больший, чем аналогичные программы, написанные на языке Ассемблер.

Однако если микроконтроллер имеет достаточно памяти и запас по быстродействию, то это увеличение программы — не проблема. **Преимуществом** же языков высокого уровня является существенное ускорение процесса разработки программы. Из всех языков высокого уровня **самым эффективным**, пожалуй, является язык СИ. Поэтому для иллюстрации языков высокого уровня мы выберем именно его.

Изучение приемов программирования мы будем осуществлять на ряде конкретных примеров:

- каждый пример будет начинаться с постановки задачи;
- затем мы научимся выбирать схемное решение;
- лишь после этого будут представлены примеры программ.

Для каждой задачи в книге приводятся два варианта программы. Одна на языке Ассемблер, вторая на языке СИ. В результате вы сможете не только научиться азам программирования на двух языках, но и понять все достоинства и недостатки каждого из языков программирования.

Все **примеры**, приведенные в моей книге, вы можете попробовать вживую на вашем компьютере. Причем текст программ не обязательно набирать вручную. Все приведенные в книге примеры вы можете скачать из Интернета с сайта **http://book.microprocessor.by.ru**. Кроме точной копии всех программ, приведенных в данной книге, на сайте вы найдете целый ряд дополнительных примеров. Все подробности смотрите на самом сайте.

1.2. Простейшая программа

Постановка задачи

Самая простая задача, которую можно придумать для микроконтроллера, может звучать следующим образом:

«Разработать устройство управления одним светодиодным индикатором при помощи одной кнопки. При нажатии кнопки светодиод должен зажечься, при отпускании — погаснуть».

С практической точки зрения это совершенно бессмысленная задача, так как для ее решения проще обойтись без микропроцессора. Но в качестве примера для обучения подойдет прекрасно.

Принципиальная электрическая схема

Попробуем разработать принципиальную электрическую схему, способную выполнять описанную выше задачу. Итак, к микроконтроллеру нам нужно подключить **светодиод** и **кнопку управления**. Как мы уже говорили, для подключения к микроконтроллеру AVR любых внешних устройств используются **порты ввода—вывода**. Причем каждый такой порт способен работать либо на ввод, либо и на вывод.

Удобнее всего светодиод подключить к одному из портов, а кнопку — к другому. В этом случае управляющая программа должна будет настроить порт, к которому подключен светодиод, на вывод, а порт, к которому подключена кнопка, на ввод. Других специальных требований к микроконтроллеру не имеется. Поэтому выберем микроконтроллер.

Очевидно, что нам нужен микроконтроллер, который имеет не менее двух портов. Данным условиям удовлетворяют многие микроконтроллеры AVR. Я предлагаю остановить свой выбор на довольно интересной **микросхеме ATiny2313**. Эта микросхема, хотя и относится к семейству «Tiny», на самом деле занимает некое промежуточное место между семейством «Tiny» и семейством «Mega». Она не так перегружена внутренней периферией и не столь сложна, как микросхемы семейства «Mega». Но и не настолько примитивна, как все остальные контроллеры семейства «Tiny».

Эта микросхема содержит два основных и один дополнительный порт ввода—вывода, имеет не только восьмиразрядный, но и шестнадцатиразрядный таймер/счетчик. Имеет оптимальные размеры (20-выводной корпус). И, по моему мнению, идеально подходит в качестве примера для изучения основ программирования. К тому же эта микросхема имеет и еще одну привлекательную особенность. По набору портов и расположению выводов она максимально приближена к микроконтроллеру AT89C2051, который был использован в качестве примера в первом издании настоящей книги.

Итак, если не считать порта А, который включается только в особом режиме, который мы пока рассматривать не будем, микроконтроллер имеет два основных порта ввода—вывода (порт В и порт D). Договоримся, что для управления светодиодом мы будем использовать младший разряд порта В (линия PB.0), а для считывания информации с кнопки управления используем младший разряд порта D (линия PD.0). Полная схема устройства, позволяющего решить поставленную выше задачу, приведена на **рис. 1.1**.

Для подключения кнопки S1 использована классическая схема. В исходном состоянии контакты кнопки разомкнуты. Через резистор R1 на вход PD.0 микроконтроллера подается «плюс» напряжения питания, что соответствует сигналу логической единицы.

При замыкании кнопки напряжение падает до нуля, что соответствует логическому нулю. Таким образом, считывая значение сигнала на соответствующем выводе порта, программа может определять момент нажатия кнопки. Несмотря на простоту данной схемы, микроконтроллер AVR позволяет ее упростить. А именно, предлагаю исключить резистор R1, заменив его внутренним нагрузочным резистором микроконтроллера. Как уже говорилось выше, микроконтроллеры серии AVR имеют встроенные нагрузочные резисторы для каждого разряда порта. Главное при написании программы — не забыть включить программным путем соответствующий резистор.

Подключение светодиода также выполнено по классической схеме. Это непосредственное подключение к выходу порта. Каждый выход микроконтроллера рассчитан на непосредственное управление светодиодом среднего размера с током потребления до 20 мА. В цепь светодиода включен токоограничивающий резистор R3.

Для того, чтобы зажечь светодиод, микроконтроллер должен подать на вывод PB.0 сигнал логического нуля. В этом случае напряжение, приложенное к цепочке R2, VD1, окажется равным напряжению питания, что вызовет ток через светодиод, и он загорится. Если же

на вывод PD.0 подать сигнал логической единицы, падение напряжения на светодиоде и резисторе окажется равным нулю, и светодиод погаснет.

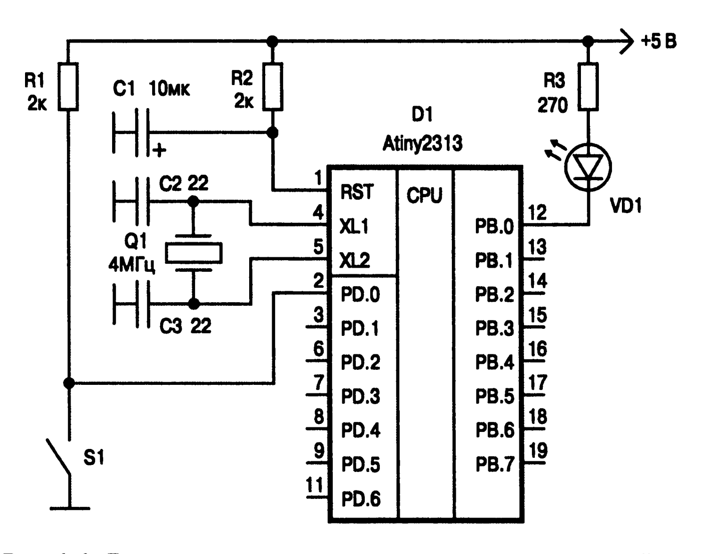


Рис. 1.1. *Принципиальная схема с одним светодиодом и одной кнопкой*

Кроме цепи подключения кнопки и цепи управления светодиодом, на схеме вы можете видеть еще несколько цепей. Это стандартные цепи, обеспечивающие нормальную работу микроконтроллера. Кварцевый резонатор Q1 обеспечивает работу встроенного тактового генератора. Конденсаторы C2 и C3 — это цепи согласования кварцевого резонатора.

Элементы C1, R2 — это стандартная цепь начального сброса. Такая цепь обеспечивает сброс микроконтроллера в момент включения питания. Еще недавно подобная цепь была обязательным атрибутом любой микропроцессорной системы. Однако технология производства микроконтроллеров достигла такого уровня, что обе эти цепи (внешний кварц и цепь начального сброса) теперь можно исключить.

Большинство микроконтроллеров AVR, кроме тактового генератора с внешним кварцевым резонатором, содержат внутренний RC-генератор, не требующий никаких внешних цепей. Если вы не предъявляете высоких требований к точности и стабильности частоты задающего генератора, то микросхему можно перевести в режим внутреннего RC-генератора и отказаться как от внешнего кварца (Q1), так и от согласующих конденсаторов (C2 и C3).

Цепь начального сброса тоже можно исключить. Любой микроконтроллер AVR имеет внутреннюю систему сброса, которая в большинстве случаев прекрасно обеспечивает стабильный сброс при включении питания. Внешние цепи сброса применяются только при наличии особых требований к длительности импульса сброса. А это бывает лишь в тех случаях, когда микроконтроллер работает в условиях больших помех и нестабильного питания.

Все описанные выше переключения производятся при помощи соответствующих fuse-переключателей. Как это можно сделать, мы увидим **в следующей главе**. Три освободившихся вывода микроконтроллера могут быть использованы как дополнительный порт (порт A). Но в данном случае в этом нет необходимости.

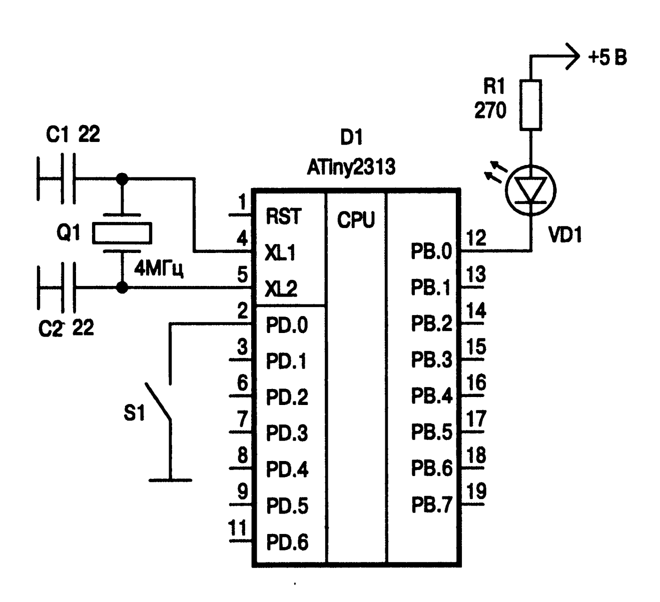


Рис. 1.2. *Усовершенствованная схема для первого задания*

Упростим схему, показанную на **рис. 1.1,** с учетом описанных выше возможностей. От внешнего кварца пока отказываться не будем. Он нам пригодиться чуть позже, когда мы начнем формировать временные интервалы. Доработанная схема изображена на **рис. 1.2**.

Алгоритм

Итак, схема у нас есть. Теперь нужно приступать к разработке программы. Разработка любой программы начинается с разработки алгоритма.

Определение. ***Алгоритм*** *— это последовательность действий, которую должен произвести наш микроконтроллер, чтобы достичь требуемого результата. Для простых задач алгоритм можно просто описать словами. Для более сложных задач алгоритм рисуется в графическом виде.*

В нашем случае алгоритм таков: После операций начальной настройки портов микроконтроллер должен войти в непрерывный цикл, в процессе которого он должен опрашивать вход, подключенный к нашей кнопке, и в зависимости от ее состояния управлять светодиодом. Опишем это подробнее.

Операции начальной настройки:

- установить начальное значение для вершины стека микроконтроллера;
- настроить порт B на вывод информации;
- подать на выход PB.0 сигнал логической единицы (потушить светодиод);
- сконфигурировать порт D на ввод;
- включить внутренние нагрузочные резисторы порта D.

Операции, составляющее тело цикла:

- прочитать состояние младшего разряда порта PD (PD.0);

- если значение этого разряда равно единице, выключить светодиод;
- если значение разряда PD.0 равно нулю, включить светодиод;
- перейти на начало цикла.

Программа на Ассемблере

Для создания программ мы используем версию Ассемблера, предложенную разработчиком микроконтроллеров AVR — фирмой Atmel. А также воспользуемся программным комплексом «AVR Studio», разработанным той же фирмой и предназначенным для создания, редактирования, трансляции и отладки программ для AVR на Ассемблере. Подробнее о программе «AVR Studio» мы поговорим в последней главе книги.

А сейчас наша **задача** — научиться создавать программы. Изучение языка будет происходить следующим образом. Я буду приводить готовый текст программы для каждой конкретной задачи, а затем подробно описывать все его элементы и объяснять, как программа работает.

Текст возможного варианта программы, реализующий поставленную выше задачу, приведен в **листинге 1.1**. Прежде, чем мы приступим к описанию данного примера, я хотел бы дать несколько общих понятий о языке Ассемблер.

Программа на Ассемблере представляет собой набор команд и комментариев (иногда команды называют инструкциями). Каждая команда занимает одну отдельную строку. Их допускается перемежать пустыми строками. Команда обязательно содержит оператор, который выглядит как имя выполняемой операции.

Некоторые команды состоят только из одного оператора. Другие же команды имеют один или два операнда (параметра). Операнды записываются в той же строке сразу после оператора, через пробел. Если операндов два, их записывают через запятую. Так, в **строке 6** нашей программы записана команда загрузки константы в регистр общего назначения. Она состоит из оператора `ldi` и двух операндов `temp` и RAMEND.

В случае необходимости перед командой допускается ставить так называемую **метку**. Она состоит из имени метки, заканчивающимся двоеточием. Метка служит для именования данной строки программы.

Затем это имя используется в различных командах для обращения к помеченной строке.

При выборе имени метки необходимо соблюдать следующие **правила**:

- имя должно состоять из одного слова, содержащего только латинские буквы и цифры;
- допускается также применять символ подчеркивания;
- первым символом метки обязательно должна быть буква или символ подчеркивания.

Строка 16 нашей программы содержит метку с именем `main`. Метка не обязательно должна стоять в строке с оператором. Допускается ставить метку в любой строке программы. Кроме команд и меток, программа содержит комментарии.

Определние. ***Комментарий*** *— это специальная запись в теле программы, предназначенная для человека. Компьютер в процессе трансляции программы игнорирует все комментарии. Комментарий может занимать отдельную строку, а может стоять в той же строке, что и команда. Начинается комментарий с символа «точка с запятой». Все, что находится после точки с запятой до конца текущей строки программы, считается комментарием.*

Если в уже готовой программе вы поставите точку с запятой в начале строки перед какой-либо командой, то данная строка для транслятора как бы исчезнет. С этого момента транслятор будет считать всю эту строку комментарием. Таким образом, можно временно отключать отдельные строки программы в процессе отладки (то есть при поиске ошибок в программе).

Кроме операторов, в языке Ассемблер применяются **псевдооператоры** или **директивы**. Если оператор — это некий эквивалент реальной команды микроконтроллера и в процессе трансляции заменяется соответствующим машинным кодом, который помещается в файл результата трансляции, то директива, хотя по форме и напоминает оператор, но не является командой процессора.

Определние. ***Директивы*** *— это специальные вспомогательные команды для транслятора, определяющие режимы трансляции и реализующие различные вспомогательные функции.*

Далее из конкретных примеров вы поймете, о чем идет речь. В данной конкретной версии Ассемблера директивы выделяются особым образом. Имя каждой директивы начинается с точки. Смотри **листинг 1.1, строки с 1 по 5.**

При написании программ на Ассемблере принято соблюдать **особую форму записи:**

- программа записывается в несколько колонок (см. **листинг 1.1**);
- аналогичные элементы разных команд принято размещать друг под другом;
- самая первая (левая) колонка зарезервирована для меток;
- если метка отсутствует, место в колонке пустует.
- следующая колонка предназначена для записи операторов
- затем идет колонка для операндов.
- оставшееся пространство (крайняя колонка справа) предназначено для комментариев.

В некоторых случаях, **например** когда текст команды очень длинный, допускается нарушать этот порядок. Но по возможности нужно оформлять программу именно так. Оформленная подобным образом программа более наглядна и гораздо лучше читается. Поэтому привыкайте писать программы правильно.

Итак, мы рассмотрели общие принципы построения программы на Ассемблере. Теперь пора приступать к подробному описанию конкретной программы, приведенной в **листинге 1.1**. И начнем мы с описания входящих в нее команд.

Директивы

.include

Присоединение к текущему тексту программы другого программного текста. Подобный прием используется практически во всех существующих языках программирования. При составлении программ часто бывает как, что в совершенно разных программах приходится применять абсолютно одинаковые программные фрагменты. Для того, чтобы не переписывать эти фрагменты из программы в программу, их принято оформлять в виде отдельного файла с таким расчетом, чтобы этот файл могли использовать все программы, где этот фрагмент потребуется.

В языке Ассемблер для присоединения фрагмента к программе используется **псевдооператор `include`**. В качестве параметра для этой директивы должно быть указано имя присоединяемого файла. Если такой оператор поставить в любом месте программы, то содержащийся в присоединяемом файле фрагмент в процессе трансляции как бы вставляется в то самое место, где находится оператор. **Например**, в программе на **листинге 1.1** в **строке 1** в основной текст программы вставляется текст из файла `tn2313def.inc`.

Кстати, подробнее об этом файле. Файл `tn2313def.inc` — это файл описаний. Он содержит описание всех регистров и некоторых других параметров микроконтроллера ATiny2313. Это описание понадобится нам для того, чтобы в программе мы могли обращаться к каждому регистру по его имени. О том, как делаются такие описания, мы поговорим при рассмотрении конкретных программ.

.list

Включение генерации листинга. В данном случае листинг — это специальный файл, в котором отражается весь ход трансляции программы. Такой листинг повторяет весь текст вашей программы, включая все присоединенные фрагменты. Против каждой строки программы, содержащей реальную команду, помещаются соответствующие ей машинные коды. Там же показываются все найденные в процессе трансляции ошибки. По умолчанию листинг не формируется. Если вам нужен листинг, включите данную команду в вашу программу.

.def

Макроопределение. Эта команда позволяет присваивать различным регистрам микроконтроллера любые осмысленные имена, упрощающие чтение и понимание текста программы. В нашем случае нам понадобится один регистр для временного хранения различных величин. Выберем для этой цели регистр `r16` и присвоим ему наименование `temp` от английского слова temporary — временный.

Данная команда выполняется в **строке 3** (см. **листинг 1.1**). Теперь в любом месте программы вместо имени r16 можно применять имя `temp`. Вы спросите: а зачем это нужно? Да для наглядности и читаемости программы. В данной программе мы будем использовать лишь один регистр, и преимущества такого переименования здесь не очень видны. Но представьте, что вы используете множество разных регистров для хранения самых разных величин. В этом случае присвоение

осмысленного имени очень облегчает программирование. Скоро вы сами в этом убедитесь. Кстати, именно таким образом определены имена всех стандартных регистров в файле `tn2313def.inc`.

.cseg

Псевдооператор выбора программного сегмента памяти. О чем идет речь? Как уже говорилось, микроконтроллер для хранения данных имеет **три вида памяти**: память программ (Flash), оперативную память (SRAM) и энергонезависимую память данных (EEPROM). Программа на Ассемблере должна работать с любым из этих трех видов памяти. Для этого в Ассемблере существует понятие «сегмент памяти». Существуют директивы, объявляющие каждый такой сегмент:

- сегмент кода (памяти программ) .... .cseg;
- сегмент данных (ОЗУ).. .dseg;
- сегмент EEPROM .... .eseg.

После объявления каждого такого сегмента он становится **текущим**. Это значит, что все последующие операторы относятся исключительно к объявленному сегменту. Объявленный сегмент будет оставаться текущим до тех пор, пока не будет объявлен какой-либо другой сегмент.

Только в сегменте кода Ассемблер описывает команды, которые затем в виде кодов будут записаны в память программ. В остальных двух сегментах используются директивы распределения памяти и директивы описания данных. Ну, к сегментам `dseg` и `eseg` мы еще вернемся. Сейчас же подробнее рассмотрим сегмент `cseg`.

Так как команды в программной памяти должны располагаться по порядку, одна за другой, то их размещение удобно автоматизировать. Программист не указывает, по какому адресу в памяти должна быть расположена та либо иная команда. Программист просто последовательно пишет команды.

А уже транслятор автоматически размещает их в памяти. Для этого используется понятие **«указатель текущего адреса»**. Указатель текущего адреса не имеет отношения к регистру адреса микроконтроллера и вообще физически не существует. Это просто понятие, используемое в языке Ассемблер. Указатель помогает транслятору разместить все команды программы по ячейкам памяти.

По умолчанию считается, что в начале программы значение текущего указателя рано нулю. Поэтому первая же команда программы будет

размещена по нулевому адресу. По мере трансляции программы указатель смещается в сторону увеличения адреса. Если команда имеет длину в один байт, то после ее трансляции указатель смещается на одну ячейку. Если команда состоит из двух байтов, на две. Таким образом, размещаются все команды программы.

.org

Принудительное позиционирование указателя текущего адреса. Иногда необходимо разместить какой-либо фрагмент программы в программной памяти не сразу после предыдущего фрагмента, а в конкретном месте программной памяти. **Например**, начиная с какого-нибудь заранее определенного адреса. Для этого используют директиву `org`.

Она позволяет принудительно изменить значение указателя текущего адреса. Оператор `org` имеет всего один параметр — новое значение указателя адреса. **К примеру**, команда `.org 0x10` установит указатель на адрес 0x10. Транслятор автоматически следит, чтобы при перемещении указателя ваши фрагменты программы не налезали друг на друга. В случае несоблюдения этого условия транслятор выдает сообщение об ошибке.

В нашей программе команда позиционирования указателя применяется всего один раз. В **строке 5** указатель устанавливается на нулевой адрес. В данном случае директива `org` имеет чисто декларативное значение, так как в начале программы значение указателя и так равно нулю.

Операторы

ldi

Загрузка в РОН числовой константы. В **строке 6** программы (Листинг 1.1) при помощи этой команды в регистр `temp` (`r16`) записывается числовая константа, равная максимальному адресу ОЗУ. Эта константа имеет имя RAMEND. Ее значение описано в файле `tn2313def.inc`. В нашем случае (для микроконтроллера ATiny2313) значение RAMEND равно $7F.

Как можно видеть из листинга 1.1, оператор `ldi` имеет два параметра:

- первый параметр — это имя РОН, куда помещается наша константа;
- второй параметр — значение этой константы.

Обратите внимание, что в команде сначала записывается приемник информации, затем ее источник. Такой же порядок вы увидите в любой другой команде, имеющей два операнда. Это **общее правило** для языка Ассемблер.

out

Вывод содержимого РОН в регистр ввода—вывода. Команда также имеет два параметра:

- первый параметр — имя РВВ, являющегося приемником информации;
- второй параметр — имя РОН, являющегося источником.

В **строке 7** программы содержимое регистра `temp` выводится в РВВ с именем `SPL`.

in

Ввод информации из регистра ввода—вывода. Имеет два параметра. Параметры те же, что и в предыдущем случае, но источник и приемник меняются местами. В **строке 19** программы содержимое регистра `PORTD` помещается в регистр `temp`.

rjmp

Команда безусловного перехода. Команда имеет всего один параметр — адрес перехода. В **строке 21** программы оператор безусловного перехода передает управление на строку, помеченную меткой `main`. То есть на **строку 19.** Данная строка демонстрирует использование метки.

На самом деле в качестве параметра оператора rjmp должен выступать так называемый относительный адрес перехода. То есть, число байт,

на которое нужно сместиться вверх или вниз от текущего адреса. Направление смещения (вверх или вниз) — это знак числа. Он определяется старшим битом. Язык Ассемблера избавляет программиста от необходимости подсчета величины смещения. Достаточно в нужной строке программы поставить метку, а в качестве адреса перехода указать ее имя, и транслятор сам вычислит значение этого параметра.

Листинг 1.1

```
;##############################################
;##                Пример 1                  ##
;##    Программа управления светодиодом       ##
;##############################################

;------------------------- Команды управления

.include "tn2313def.inc"          ; Присоединение файла описаний
.list                             ; Включение листинга

.def   temp = R16                 ; Определение главного рабочего регистра

;------------------------- Начало программного кода

          .cseg                   ; Выбор сегмента программного кода
          .org    0               ; Установка текущего адреса на ноль

;-------------------------- Инициализация стека

          ldi     temp, RAMEND    ; Выбор адреса вершины стека
          out     SPL, temp       ; Запись его в регистр стека

;-------------------------- Инициализация портов ВВ

          ldi     temp, 0         ; Записываем 0 в регистр temp
          out     DDRD, temp      ; Записываем этот ноль в DDRD (порт PD на ввод)

          ldi     temp, 0xFF      ; Записываем число $FF в регистр temp
          out     DDRB, temp      ; Записываем temp в DDRB (порт PB на вывод)
          out     PORTB, temp     ; Записываем temp в PORTB (потушить светодиод)
          out     PORTD, temp     ; Записываем temp в PORTD (включаем внутр. резист.)

;--------------------------- Инициализация компаратора

          ldi     temp, 0x80      ; Выключение компаратора
          out     ACSR, temp

;------------------------- Основной цикл

main:     in      temp, PIND      ; Читаем содержимое порта PD
          out     PORTB, temp     ; Пересылаем в порт PB
          rjmp    main            ; К началу цикла
```

При использовании команды `rjmp` существует одно **ограничение**. Соответствующая команда микроконтроллера кодируется при помощи одного шестнадцатиразрядного слова. Для указания величины смещения она использует всего двенадцать разрядов. Поэтому такая команда может вызвать переход в пределах ±2 Кбайт. Если вы расположите метку слишком далеко от оператора `rjmp`, то при трансляции программы это вызовет сообщение об ошибке.

Описание программы (листинг 1.1)

Текст программы **начинается** шапкой с названием программы. **Шапка** представляет собой несколько строк комментариев. Шапка в начале программы помогает отличать программы друг от друга. Кроме названия программы, в шапку можно поместить ее версию, а также дату написания.

Самая первая команда программы — это псевдокоманда `include`, которая присоединяет к основному тексту программы файл описаний (см. листинг 1.1 **строка 1**). В стандартном пакете AVR-Studio имеется целый набор подобных файлов описаний. Для каждого микроконтроллера серии AVR свой отдельный файл. Все стандартные файлы описаний находятся в директории «C:\Program Files\Atmel\AVR Tools\AvrAssembler\Appnotes\». Программисту нужно лишь выбрать нужный файл и включить подобную **строку** в свою программу. Учтите, что без присоединения файла описаний дальнейшая программа работать не будет.

Для микроконтроллера ATiny2313 **файл описаний** имеет название `tn2313def.inc`. Если файл описаний находится в указанной выше директории, то в команде `include` достаточно лишь указать его полное имя (с расширением). Указывать полный путь необязательно.

Назначение команды `.list` (**строка 2**), надеюсь, у вас уже не вызывает вопросов. Остановимся на **команде макроопределения (строка 3)**. Эта команда, как уже говорилось, присваивает регистру `r16` имя `temp`. Дальше в программе регистр `temp` используется для временного хранения промежуточных величин. Уместно задаться **вопросом**: почему выбран именно `r16`, а, к примеру, не `r0`? Это становится понятно, если вспомнить, что регистры, начиная с `r0` и заканчивая `r15`, имеют меньше возможностей. **Например**, в **строке 14** программы регистр `temp` используется в команде `ldi`. Однако команда `ldi` не работает с регистрами `r0–r15`. Именно по этой причине мы и выбрали `r16`.

Следующие две команды (**строки 4, 5**) подробно описаны в начале этого раздела. Они служат для выбора программного сегмента памяти и установки начального значения указателя.

В **строках 6 и 7** производится **инициализация стека**. В регистр стека SPL записывается **адрес его вершины**. В качестве адреса выбран самый верхний адрес ОЗУ. Для обозначения этого адреса в данной версии Ассемблера существует специальная константа с именем RAMEND. Значение этой константы определяется в файле описаний (в нашем случае в файле tn2313def.inc). Для микроконтроллера Atiny2313 константа RAMEND равна 0xDF.

Одной строкой записать константу в регистр стека невозможно, так как в системе команд микроконтроллеров AVR отсутствует подобная команда. Отсутствующую команду мы заменяем двумя другими. И тут нам пригодится **регистр temp**. Он послужит в данном случае передаточным звеном. Сначала **константа RAMEND** помещается в регистр temp (**строка 6**), а затем уже содержимое temp помещается в **регистр SPL** (**строка 7**).

В **строках 8—12** производится **настройка портов ввода—вывода**. Ранее мы уже договорились, что порт PD у нас будет работать на ввод, а порт PB — на вывод. Для выбора нужного направления передачи информации запишем управляющие коды в соответствующие регистры DDRx. Во все разряды **регистра DDRD** запишем нули (настройка порта PD на ввод), а во все разряды регистра DDRB запишем единицы (настройка порта PB на вывод). Кроме того, нам нужно включить внутренние нагрузочные резисторы порта PD. Для этого мы запишем единицы (то есть число 0xFF) во все разряды **регистра PORTD**. И, наконец, в момент старта программы желательно погасить светодиод. Для этого мы запишем единицы в разряды порта PB.

Все описанные выше действия по настройке порта также выполняются с использованием промежуточного **регистра temp**. Сначала в него помещается ноль (**строка 8**). Ноль записывается только в регистр DDRD (**строка 9**). Затем в регистр temp помещается число 0xFF (**строка 10**). Это число по очереди записывается в регистры DDRB, PORTB, PORTD (**строки 11, 12, 13**).

Строки 14 и 15 включены в программу для перестраховки. Дело в том, что встроенный компаратор микроконтроллера после системного сброса остается включен. И хотя прерывания при этом отключены и срабатывание компаратора не может повлиять на работу нашей программы, мы все же отключим компаратор. Именно это и делается в **строках 14** и **15**.

Здесь уже знакомым нам способом с использованием регистра `temp` производится запись константы 0x80 в регистр ACSR. **Регистр ACSR предназначен** для управления режимами работы компаратора, а константа 0x80, записанная в этот регистр, отключает компаратор.

Настройкой компаратора заканчивается подготовительная часть программы. Подготовительная часть занимает **строки** 1—15 и выполняется всего один раз после включения питания или после системного сброса.

Строки 16—18 составляет **основной цикл программы.**

Определение. *Основной цикл — это часть программы, которая повторяется многократно и выполняет все основные действия.*

В нашем случае, согласно алгоритму, действия программы состоят в том, чтобы прочитать состояние кнопки и перенести его на светодиод. Есть много способов перенести содержимое младшего разряда порта PD в младший разряд порта PB. В нашем случае реализован самый простой вариант. Мы просто переносим одновременно все разряды. Для этого достаточно двух операторов.

Первый из них читает содержимое порта PD и запоминает это содержимое в **регистре `temp` (строка 16)**. Следующий оператор записывает это число в **порт PB (строка 17)**. Завершает основной цикл программы **оператор безусловного перехода (строка 18)**. Он передает управление **по метке `main`**.

В результате три оператора, составляющие тело цикла, повторяются бесконечно. Благодаря этому бесконечному циклу все изменения порта PD тут же попадают в порт PB. По этой причине, если кнопка S1 не нажата, логическая единица со входа PD0 за один проход цикла передается на выход PB0. И светодиод не светится. При нажатии кнопки S1 логический ноль со входа PD0 поступает на выход PB0, и светодиод загорается.

Эта же самая программа без каких-либо изменений может обслуживать до семи кнопок и такое же количество светодиодов. Дополнительные кнопки подключаются к линиям PD1—PD6, а дополнительные светодиоды (каждый со своим токоограничивающим резистором) к выходам PB1—PB7. При этом каждая кнопка будет управлять своим собственным светодиодом. Такое стало возможным потому, что все выводы каждого из двух портов мы настроили одинаково (смотри **строки 8—13**).

Программа на языке СИ

Для создания программ на языке СИ мы будем использовать **программную среду CodeVisionAVR**. Это среда специально предназначена для разработки программ на языке СИ для микроконтроллеров серии AVR. Среда CodeVisionAVR не имеет своего отладчика, но позволяет отлаживать программы, используя возможности системы AVR Studio.

Отличительной **особенностью** системы CodeVisionAVR является наличие мастера-построителя программы. **Мастер-построитель** облегчает работу программисту. Он избавляет от необходимости листать справочник и выискивать информацию о том, какой регистр за что отвечает и какие коды нужно в него записать. **Результат** работы мастера — это заготовка будущей программы, в которую включены все команды предварительной настройки, а также заготовки всех процедур языка СИ. Поэтому давайте сначала научимся работать с мастером, создадим заготовку нашей программы на языке СИ, а затем разберем подробнее все ее элементы. И уже после этого создадим из заготовки готовую программу.

Для того, чтобы понять работу мастера, нам необходимо прояснить **несколько новых понятий**. Программа CodeVisionAVR, как и любая современная среда программирования, работает не просто с текстом программы, а с так называемым **Проектом.**

Определение. ***Проект****, кроме текста программы, содержит ряд вспомогательных файлов, содержащих дополнительную информацию, такую, как тип используемого процессора, тактовую частоту кварца и т. п. Эта информация необходима в процессе трансляции и отладки программ. За исключением текста программы, все остальные файлы проекта создаются и изменяются автоматически.*

Задача программиста лишь написать текст программы, для которого в проекте отводится отдельный файл с расширением «с». Например, «proba.c».

При помощи мастера мы будем создавать новый проект. В дальнейшем мы еще рассмотрим подробно процесс установки и работу с программной средой CodeVisionAVR. Сейчас же считаем, что она установлена и запущена.

Для создания нового проекта выберем в меню «File» пункт «New». Откроется небольшой диалог, в котором вы должны выбрать тип создаваемого файла. Предлагается два варианта:

- «Source» (Исходный текст программы);
- «Project» (Проект).

Выбираем Project и нажимаем «Ok». Появляется окно с вопросом «You are about to create a new project. Do you want to use the CodeWizardAVR?». В переводе на русский это означает: «Вы создаете новый проект. Будете ли вы использовать построитель CodeWizardAVR?». Мы договорились, что будем, поэтому выбираем «Yes», после чего открывается окно построителя (см. **рис. 1.3**). Как видите, это окно имеет множество вкладок, каждая из которых содержит элементы выбора режимов.

Все эти управляющие элементы позволяют **настроить параметры создаваемой заготовки программы**. Сразу после запуска мастера все параметры принимают значения по умолчанию (все внутренние устройства выключены, а все порты ввода—вывода настроены на ввод, внутренние нагрузочные резисторы отключены). Это соответствует начальному состоянию микроконтроллера непосредственно после системного сброса.

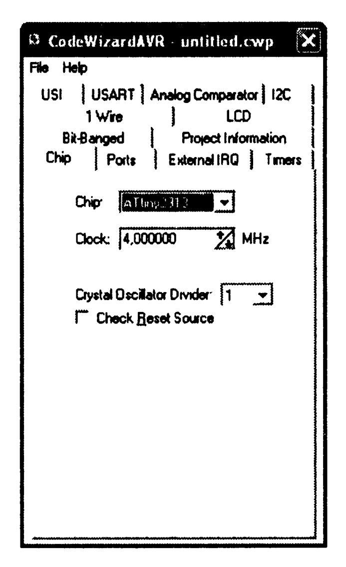


Рис. 1.3. *Окно мастера CodeVisionAVR*

Пройдемся немного по **вкладкам мастера** и выберем необходимые нам параметры. Те же параметры, которые нам не нужны, мы трогать пока не будем (оставим значения по умолчанию).

Первая вкладка называется «Chip». На этой вкладке мы можем выбрать общие параметры проекта. Используя выпадающий список «Chip», выберем тип микроконтроллера. Для этого щелкаем мышью по окошку и в выпавшем списке выбираем ATiny2313.

При помощи поля «Clock» выбираем частоту кварцевого резонатора. В нашем случае она должна быть равна 4 МГц. При помощи поля «Crystal Oscillator Divider» выбирается **коэффициент деления тактового генератора**. Этот параметр требует пояснений. Дело в том, что выбранный нами микроконтроллер имеет систему предварительного деления тактовых импульсов. Если частота тактового генератора нас не устраивает, мы можем поделить ее, и микроконтроллер будет работать на другой, более низкой частоте. Коэффициент деления может изменяться от 1 до 256. Мы выберем его равным единице (без деления). То есть оставим значение по умолчанию.

Без изменений оставим флажок «Check Reset Source» (Проверка источника сигнала сброса). Не будем углубляться в его назначение. При желании вы все поймете, прочитав [4]. Достаточно будет понять, что включение данного флажка добавляет к создаваемой программе процедуру, связанную с определением источника сигнала системного сброса.

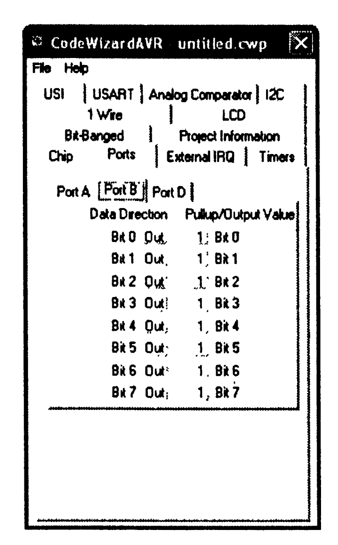


Рис. 1.4. *Настройка порта PB*

Покончив с общими настройками, перейдем к **вкладке (Порты)**. Эта вкладка позволяет настроить все имеющиеся порты ввода—вывода. На вкладке «Ports» мы видим еще три вкладки поменьше (см. **рис. 1.4**). По одной вкладке для каждого порта. Как уже говорилось выше, порт PA в данной схеме мы не применяем. Поэтому сразу выбираем вкладку «Port B».

На вкладке мы видим два столбца с параметрами. Столбец **«Data direction» (Направление передачи данных)** позволяет настроить каждую линию порта либо на ввод, либо на вывод. По умолчанию каждый параметр имеет значение «In» (вход). Поменяем для каждого разряда это значение на «Out» (Выход).

Для того, чтобы поменять значение разряда, достаточно щелкнуть по полю с надписью «In» один раз мышью, и параметр сменится на «Out». Повторный щелчок заставит вернуться к «In». Каждое поле столбца «Data direction» определяет, какое значение будет присвоено соответствующему разряду регистра DDRB в нашей будущей программе.

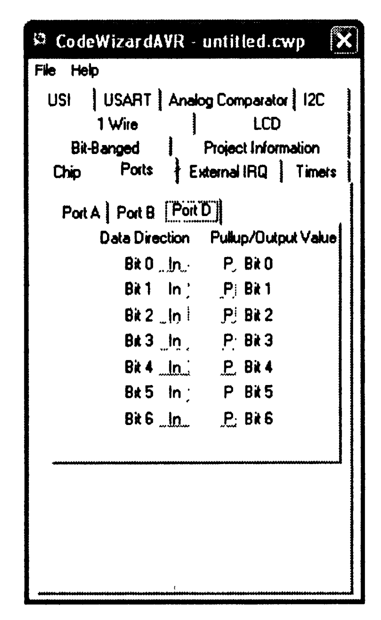


Рис. 1.5. *Настройка порта PD*

Второй столбец на той же вкладке **называется «Pullup / Output Value» (Включение нагрузки / Выходное значение)**. Этот столбец

определяет, какое значение будет присвоено каждому из разрядов регистра PORTB. В нашем случае порт PB работает на вывод. Поэтому содержимое регистра PORTB определяет выходное значение всех разрядов порта. По умолчанию все они имеют значение «0». Но по условиям нашей задачи они должны быть равны единице (при старте программы светодиод должен быть отключен). Поэтому изменим все нули на единицы. Для этого также достаточно простого щелчка мыши. **В рез**ультате всех операций вкладка «Port B» будет выглядеть так, как это показано на **рис. 1.4**.

Теперь перейдем к **настройке последнего порта**. Для этого выберем вкладку «Potr D». На вкладке мы увидим такие же органы управления, как на вкладке «Port B» (см. **рис. 1.5**). По условиям задачи порт PD микроконтроллера должен работать на ввод. Поэтому состояние элементов первого столбца мы не меняем.

Однако не забывайте, что нам нужно включить внутренние нагрузочные резисторы для каждого из входов. Для этого изменим значения элементов второго столбца. Так как порт PD работает в режиме ввода, элементы в столбце «Pullup / Output Value» принимают значение «T» или «P».

«T» (Terminate) означает отсутствие внутренней нагрузки, а **«P» (Pull-up)** означает: нагрузка включена. Включим нагрузку для каждого разряда порта PD, изменив при помощи мыши значение поля с «T» на «P». В результате элементы управления будут выглядеть так, как показано на **рис. 1.5**.

Для данной простейшей программы настройки можно считать оконченными. Остальные системы микроконтроллера нам пока не нужны. Оставим их настройки без изменений.

Воспользуемся еще одним **полезным свойством мастера программ**. Откроем вкладку «Project Information» (см. **рис. 1.6**). В поле «Project Name» вы можете занести название вашего проекта. **Поле «Version»** предназначено для номера версии. В **поле «Date»** помещают дату разработки программы. В **полях «Author»** и **«Company»** помещается, соответственно, имя автора и название компании. В **поле «Comments:»** можно поместить любые необходимые комментарии. Вся эта информация будет автоматически помещена в заголовок будущей программы.

CodeWizardAVR - untitled.cwp
File Help
USI | USART | Analog Comparator | I2C
1 Wire | LCD
Chip | Ports | External IRQ | Timers
Bit-Banged | Project Information
Project Name.
Prog1
Version 1 Date 6 01 2006
Author. Belov
Company. Home
Comments.
Пример 1
Управление одним светодиодом

Рис. 1.6. *Занесение информации для заголовка программы*

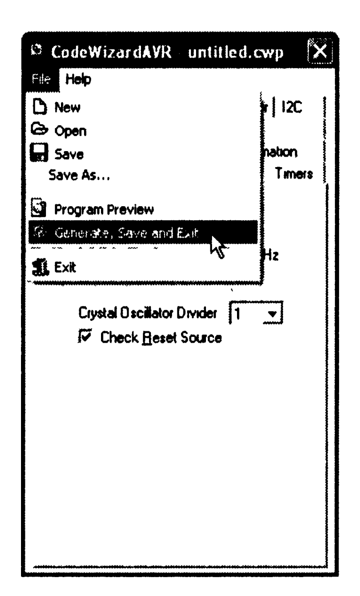


Рис. 1.7. *Запуск процесса генерации программы*

После того, как все параметры выставлены, приступаем непосредственно **к процессу генерации программы**. Для этого выбираем в меню «File» нашего мастера пункт «Generate, Save and Exit», как это показано на **рис. 1.7**. Процесс генерации начнется с запроса имени файла будущей программы. Для этого откроется стандартный диалог сохранения файла, в котором вы сначала должны выбрать каталог, а затем указать имя файла программы. Что касается каталога, то вам нужно самостоятельно выбрать каталог, где будет размещен весь ваш проект.

Рекомендуется для каждого проекта создавать свой отдельный каталог. В нашем случае самое простое — назвать каталог именем «Prog1». А вернее, выбрать что-то вроде «C:\AVR\C\Prog1\». Если каталог еще не создан, то вы можете создать его прямо в диалоге создания файла. Файлу рекомендую присвоить то же самое имя: «Prog1».

Однако вы можете выбрать имя по своему усмотрению. Для завершения процесса создания файла нажмите кнопку «Сохранить». В результате на ваш жесткий диск запишется файл «Prog1.c», в который будет помещен созданный построителем текст заготовки вашей программы. Расширение «.c» набирать не нужно. Оно присваивается автоматически. Это стандартное расширение для программ на языке СИ.

Однако процесс генерации проекта на этом не заканчивается. Сразу после того, как файл программы будет записан, откроется новый диалог, который запросит имя для файла проекта. **Файл проекта** предназначен для хранения параметров конкретного проекта.

Кроме типа используемой микросхемы и частоты задающего генератора, файл проекта используется для хранения вспомогательной информации, такой как наличие и расположение точек останова, закладок и т. д. Подробнее о закладках и точках останова мы узнаем, когда будем изучать работу отладчика (см. **раздел 5.1.5**). В качестве имени файла проекта удобнее всего использовать то же самое имя, что и для текста программы. То есть «Prog1». Файлу проекта автоматически присваивается расширение «.prj».

Когда файл проекта будет записан, диалог записи файла откроется в третий раз. Теперь нам предложат **выбрать имя для файла данных построителя**. В этот файл будут записаны текущие значения всех параметров, мастера. То есть значения всех управляющих элементов со всех его вкладок. И те, которые мы изменили в процессе настройки, и те, которые остались без изменений (по умолчанию).

Эти данные могут понадобиться, если потребуется заново пересоздать проект. Используя файл данных построителя, вы можете в любой

момент восстановить значения всех его элементов, немного подкорректировать их и создать новый проект. Файлу данных построителя присвоим то же самое имя, что обоим предыдущим («Prog1»). Новый файл получит расширение «.cwp» (Code Wizard Project).

После того, как и этот файл будет записан, процесс генерации проекта завершается. На экране появляются два новых окна. В одном окне открывается **содержимое файла «Prog1.c»**. Втрое окно — это **файл комментариев**. Сюда вы можете записать, а затем сохранить на диске любые замечания по вашей программе. В дальнейшем они всегда будут у вас перед глазами.

Посмотрим теперь, что же сформировал наш построитель. Текст программы, полученной описанным выше образом, дополненный русскоязычными комментариями, приведен в **листинге 1.2**.

Все русскоязычные комментарии дублируют соответствующие англоязычные комментарии, автоматически созданные в процессе генерации программы. Кроме новых комментариев, в программу добавлена только одна дополнительная строка (**строка 32**). Именно она превращает созданную построителем заготовку в законченную программу. **Итак, мы получили программу на языке СИ.**

Работа программы, написанной на языке Си

Теперь наша задача — разобраться, как программа работает. Именно этим мы сейчас и займемся. Но сначала небольшое введение в новый для нас язык СИ.

Программа на языке **СИ, в отличие от Ассемблера,** гораздо более абстрагирована от системы команд микроконтроллера. Основные операторы языка СИ вовсе не привязаны к командам микроконтроллера. Для реализации всего одной команды на языке СИ на самом деле используется не одна, а несколько команд микроконтроллера. Иногда даже целая небольшая программа.

В результате облегчается труд программиста, так как он теперь работает с более крупными категориями. Ему не приходится вдаваться в мелкие подробности, и он может сосредоточиться на главном. Язык СИ так же, как и другие языки программирования, состоит из команд. Для записи каждой команды СИ использует свои операторы и псевдооператоры. Но форма написания команд в программе приближена к форме, принятой в математике. Сейчас вы в этом убедитесь сами.

Листинг 1.2

```
/*****************************************************
This program was produced by the
CodeWizardAVR V1.24.4 Standard
Automatic Program Generator
© Copyright 1998-2004 Pavel Haiduc, HP InfoTech s.r.l
http://www.hpinfotech.com
e-mail:office@hpinfotech.com

Project : Prog1
Version : 1
Date    : 06.01.2006
Author  : Belov
Company : Home
Comments:
Пример 1
Управление светодиодом


Chip type           : ATtiny2313
Clock frequency     : 4,000000 MHz
Memory model        : Tiny
External SRAM size  : 0
Data Stack size     : 32
*****************************************************/

1   #include <tiny2313.h>

    // Declare your global variables here (определение ваших глобальных переменных)

2   void main(void)
    {
    // Crystal Oscillator division factor: 1
    // Коэффициент деления частоты системного генератора: 1
3   CLKPR=0x80;
4   CLKPR=0x00;

    // Input/Output Ports initialization (Инициализация портов ввода-вывода)
    // Port A initialization  (Инициализация порта A)
    // Func2=In Func1=In Func0=In
    // State2=T State1=T State0=T
5   PORTA=0x00;
6   DDRA=0x00;

    // Port B initialization (Инициализация порта B)
    // Func7=Out Func6=Out Func5=Out Func4=Out Func3=Out Func2=Out Func1=Out Func0=Out
    // State7=1 State6=1 State5=1 State4=1 State3=1 State2=1 State1=1 State0=1
7   PORTB=0xFF;
8   DDRB=0xFF;

    // Port D initialization (Инициализация порта D)
    // Func6=In Func5=In Func4=In Func3=In Func2=In Func1=In Func0=In
    // State6=P State5=P State4=P State3=P State2=P State1=P State0=P
9   PORTD=0x7F;
10  DDRD=0x00;

    // Timer/Counter 0 initialization (Инициализация таймера/счетчика 0)
    // Clock source: System Clock  (Источник сигнала: системный генератор)
    // Clock value: Timer 0 Stopped  (Значение частоты: Таймер 0 остановлен)
    // Mode: Normal top=FFh   (Режим Normal  макс. значение FFH)
    // OC0A output: Disconnected   (Выход OC0A отключен)
    // OC0B output: Disconnected   (Выход OC0B отключен)
11  TCCR0A=0x00;
12  TCCR0B=0x00;
13  TCNT0=0x00;
14  OCR0A=0x00;
15  OCR0B=0x00;

    // Timer/Counter 1 initialization (Инициализация таймера/счетчика 1)
    // Clock source: System Clock   (Источник сигнала: системный генератор)
    // Clock value: Timer 1 Stopped  (Значение частоты: Таймер 1 остановлен)
    // Mode: Normal top=FFFFh   (Режим Normal  макс. значение FFFFH)
    // OC1A output: Discon.   (Выход OC0A отключен)
    // OC1B output: Discon.   (Выход OC0B отключен)
    // Noise Canceler: Off
```

```
// Input Capture on Falling Edge
TCCR1A=0x00;
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
ICR1H=0x00;
ICR1L=0x00;
OCR1H=0x00;
OCR1L=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// External Interrupt(s) initialization (Инициализация внешних прерываний)
// INT0: Off   (Прерывание INT0 выключено)
// INT1: Off   (Прерывание INT1 выключено)
GIMSK=0x00;
MCUCR=0x00;

// Timer(s)/Counter(s) Interrupt(s) initialization
// (Инициализация прерываний от таймеров)
TIMSK=0x00;

// Universal Serial Interface initialization
// (Инициализация универсального последовательного интерфейса)
// Mode: Disabled   (Режим:  Выключен)
// Clock source: Register & Counter=no clk.
// USI Counter Overflow Interrupt: Off
USICR=0x00;

// Analog Comparator initialization (Инициализация аналогового компаратора)
// Analog Comparator: Off   (Аналоговый компаратор:  Выключен)
// Analog Comparator Input Capture by Timer/Counter 1: Off
ACSR=0x80;

while (1)
         {
                 // Place your code here  (Пожалуйста вставьте ваш код)
                 PORTB=PIND;
         };
}
```

В языке СИ для хранения различных данных используются **переменные**. Понятие «переменная» в языке СИ аналогично одноименному математическому понятию. Каждая переменная имеет свое имя, и ей можно присваивать различные значения. Используя переменные, можно строить различные выражения. Каждое выражения представляет собой одну или несколько переменных и числовых констант, связанных арифметическими и (или) логическими операциями. **Например:**

a*b — произведение переменных a и b (символ * означает умножение);

k1/2 — переменная k1, деленная на два (символ «/» означает деление);

massa1 + massa2 — сумма двух переменных (massa1 и massa2);

tkon << 2 — циклический сдвиг содержимого переменной tkon на 2 разряда влево;

dat & mask — логическое умножение (операция «И») между двумя переменными (dat и mask).

Приведенные примеры — это простейшие выражения, каждое из которых состоит всего их двух членов. Язык СИ допускает выражения практически любой сложности.

В языке СИ **переменные делятся на типы**. Переменная каждого типа может принимать значения из одного определенного диапазона (см. **табл. 1.1**). **Например**:

- переменная типа **char** — это только целые числа;
- переменная типа **float** — вещественные числа (десятичная дробь) и т. д.

Использование переменных нескольких фиксированных типов — это отличительная особенность любого языка высокого уровня. Разные версии языка СИ поддерживают различное количество типов переменных. Версия СИ, используемая в CodeVisionAVR, поддерживает тринадцать типов переменных (см. **табл. 1.1**).

Таблица 1.1. *Типы данных языка СИ*

Название	Количество бит	Значение
bit	1	0 или 1
char	8	-128 — 127
unsigned char	8	0 — 255
signed char	8	-128 — 127
int	16	-32768 — 32767
short int	16	-32768 — 32767
unsigned int	16	0 — 65535
signed int	16	-32768 — 32767
long int	32	-2147483648 — 2147483647
unsigned long int	32	0 — 4294967295
signed long int	32	-2147483648 — 2147483647
float	32	±1.175e-38 — ±3.402e38
double	32	±1.175e-38 — ±3.402e38

В языке СИ любая переменная, прежде чем она будет использована, должна быть описана. При **описании** задается ее тип. В дальнейшем диапазон принимаемых значений должен строго соответствовать выбранному типу переменной. Описание переменной и задание ее типа необходимы потому, что оттранслированная с языка СИ программа выделяет для хранения значений каждой переменной определенные ресурсы памяти.

Это могут быть ячейки ОЗУ, регистры общего назначения или даже ячейки EEPROM или Flash-памяти (памяти программ). В зависимости от заданного типа, выделяется различное количество ячеек для каждой конкретной переменной. Косвенно о количестве выделяемых ячеек можно судить по содержимому графы «Количество бит» **табл. 1.1**. Описывая переменную, мы сообщаем транслятору, сколько ячеек выделять и как затем интерпретировать их содержимое. Посмотрим, как выглядит строка описания переменной в программе. Она представляет собой запись следующего вида:

Тип Имя;

где «Тип» — это тип переменной, а «Имя» — ее имя.

Имя переменной выбирает программист. Принцип формирования имен в языке СИ не отличается от подобного принципа в языке Ассемблер. Допускается использование только латинских букв, цифр и символа подчеркивания. Начинаться имя должно с буквы или символа подчеркивания.

Кроме арифметических и логических выражений язык СИ использует **функции**.

Определение. ***Функция*** *в языке СИ — это аналог соответствующего математического понятия. Функция получает одно или несколько значений в качестве параметров, производит над ними некие вычисления и возвращает результат.*

Правда, в отличие от математических функций, функции языка СИ не всегда имеют входные значения и даже не обязательно возвращают результат. Далее на конкретных примерах мы увидим, как и почему это происходит.

Вообще, роль функций в языке СИ огромная. Программа на языке СИ просто-напросто состоит из одной или нескольких функций. Каждая функция имеет свое имя и описание. По имени производится обращение к функции. Описание определяет выполняемые функцией действия и преобразования. Вот как выглядит описание функции в программе СИ:

```
тип Name (список параметров)
        {
        тело функции
        }
```

Здесь **Name** — это **имя функции**. Имя для функции выбирается по тем же правилам, что и для переменной. При описании функции перед ее именем положено указать тип возвращаемого значения. Это необходимо транслятору, так как для возвращаемого значения он тоже резервирует ячейки.

Если перед именем функции вместо типа возвращаемого значения записать слово `void`, то это будет означать, что данная функция не возвращает никаких значений. В круглых скобках после имени функции записывается список передаваемых в нее параметров.

Функция может иметь любое количество параметров. Если параметров два и более, то они записываются через запятую. Перед именем каждого параметра также должен быть указан его **тип**. Если у функции нет параметров, то в скобках вместо списка параметров должно стоять **слово `void`**. В фигурных скобках размещается **тело функции**.

Определение. *Тело функции — это набор операторов на языке СИ, выполняющих некие действия. В конце каждого оператора ставится точка с запятой. Если функция небольшая, то ее можно записать в одну* ***строку***.

В этом случае операторы, составляющие тело функции, разделяет только точка с запятой. Вот **пример** такой записи:

тип `Name` *(список параметров) { тело функции }*

Любая программа на языке СИ должна обязательно содержать **одну главную функцию**. Главная функция должна иметь **имя `main`**. Выполнение программы всегда начинается с выполнения функции `main`. В нашем случае (см. **листинг 1.2**) описание функции `main` начинается со **строки 2** и заканчивается в конце программы. Функция `main` в данной версии языка СИ никогда не имеет параметров и никогда не возвращает никакого значения.

Тело функции, кроме команд, может содержать описание переменных. Все переменные должны быть описаны в самом начале функции, до первого оператора. Такие переменные могут быть использованы только в той функции, в начале которой они описаны. Вне этой функции данной переменной как бы не существует.

Если вы объявите переменную в одной функции, а примените ее в другой, то транслятор выдаст **сообщение об ошибке**. Это дает возмож-

ность объявлять внутри разных функций переменные с одинаковыми именами и использовать их независимо друг от друга.

Определение. *Переменные, объявленные внутри функций, называются **локальными**. При написании программ иногда необходим другой порядок использования переменных. Иногда нужны переменные, которые могут работать сразу со всеми функциями. Такие переменные называются **глобальными переменными**.*

Глобальная переменная объявляется не внутри функций, а в начале программы, еще до описания самой первой функции. Не спешите без необходимости делать переменную глобальной. Если программа достаточно большая, то можно случайно присвоить двум разным переменным одно и то же имя, что приведет к ошибке. Такую ошибку очень трудно найти.

Для того, чтобы все вышесказанное было понятнее, обратимся к конкретному примеру — программе (**листинг 1.2**). А начнем мы изучение этой программы с описания нам пока неизвестных используемых там команд.

include

Оператор присоединения внешних файлов. Данный оператор выполняет точно такую же роль, что и аналогичный оператор в языке Ассемблер. В **строке 1** программы (**листинг 1.2**) этот оператор присоединяет к основному тексту программы стандартный текст описаний для микроконтроллера ATtiny2313.

while

Оператор цикла. Форма написания команды `while` очень похожа на форму описания функции. В общем случае команда while выглядит следующим образом:

```
while (условие)
      {
      тело цикла
      };
```

Перевод английского слова `while` — «пока». Эта команда организует цикл, многократно повторяя тело цикла до тех пор, пока выполняется «условие», то есть пока выражение в скобках является истинным. В языке СИ принято считать, что выражение истинно, если оно не равно нулю, и ложно, если равно.

Определение. *Тело цикла — это ряд любых операторов языка СИ. Как и любая другая команда, вся конструкция `while` должна заканчиваться символом «точка с запятой».*

В программе на **листинге 1.2** оператор `while` вы можете видеть в **строке 31**. В качестве условия в этом операторе используется просто число 1. Так как 1 — не ноль, то такое условие всегда истинно. Такой прием позволяет создавать бесконечные циклы. Это значит, что цикл, начинающийся в **строке 31**, будет выполняться бесконечно. Тело цикла составляет единственная команда (**строка 32**).

Комментарии

В программе на языке СИ так же, как и в Ассемблере, широко используются комментарии. В языке СИ принято два способа написания комментариев. **Первый способ** — использование специальных обозначений начала и конца комментария. Начало комментария помечается парой символов /*, а конец комментария символами */. Это выглядит следующим образом:

/* Комментарий */

Причем комментарий, выделенный таким образом, может занимать не одну, а несколько строк. В **листинге 1.2** шапка программы выполнена в виде комментария, который записан именно таким образом. **Второй способ** написания комментария — двойная наклонная черта (//).

В этом случае комментарий начинается сразу после двойной наклонной черты и заканчивается в конце текущей строки. В **листинге 1.2** такой способ применяется по всему тексту программы.

Описание программы (листинг 1.2)

Как уже говорилось, текст программы, который вы видите в **листинге 1.2**, в основном сформирован автоматически. Большую часть программы занимает **функция main**. Она начинается в **строке 2** и заканчивается в конце программы. Вся программа снабжена подробными комментариями, которые также сформированы автоматически.

Исключения составляют все русскоязычные комментарии, которые я добавил вручную, и одна **строка** в конце программы (**строка 32**). Начинается программа с заголовка. В начале заголовка мастер поместил информацию о том, что программа создана при помощи CodeWizardAVR. Далее в заголовок включен блок информации из вкладки «Project Information». Эту информацию мы с вами набирали собственноручно. Далее заголовок сообщает тип процессора, его тактовую частоту, модель памяти (Tiny — означает малая модель), размер используемой внешней памяти и размер стека.

В **строке 1** находится **команда include**, присоединяющая файл описаний. После команды include мастер поместил сообщение для программиста. Сообщение предупреждает о том, что именно в этом месте программисту нужно поместить описание всех глобальных переменных (если, конечно, они вам понадобятся). В данном конкретном случае глобальные переменные нам не нужны. Поэтому мы добавлять их не будем.

Теперь перейдем к **функции main**. Функция `main` содержит в себе набор команд, настройки системы (**строки 3—30**) и заготовку главного цикла программы (**строка 31**).

Определение. ***Настройка системы*** *— это запись требуемых значений во все управляющие регистры микроконтроллера.*

В программе на Ассемблере (**листинг 1.1**) мы тоже производили подобную настройку. Однако там мы ограничились инициализацией портов ввода—вывода и отключением компаратора. Состояние всех остальных служебных регистров микроконтроллера программа на Ассемблере не меняла. То есть оставляла значения по умолчанию.

На языке СИ можно было бы поступить точно так же. Но мастер-построитель программы поступает по-другому. Он присваивает значения всем без исключения служебным регистрам. И тем, значения которых должны отличаться от значений по умолчанию, и тем, значения которых не изменяются.

В последнем случае регистру присваивается то же самое значение, какое он и так имеет после системного сброса. Такие, на первый взгляд, избыточные действия имеют свой смысл. Они гарантируют правильную работу программы в том случае, если в результате ошибки в программе управление будет передано на ее начало.

Лишние команды при желании можно убрать. В нашем случае достаточно оставить лишь команды инициализации портов (**строки 7, 8** для порта PB и **строки 9, 10** для порта PD). А также команду инициализации компаратора (**строка 30**).

Теперь посмотрим, как же **происходит присвоение значений**. В **строке 3** регистру CLKPR присваивается значение 0x80. Для присвоения значения используется хорошо знакомый нам символ «=» (равно). В языке СИ такой символ называется **оператором присвоения**. Таким же самым образом присваиваются значения и всем остальным регистрам.

Что касается настройки портов PB и PD, то в **строках 7, 8** регистрам PORTB и DDRB присваивается значение 0x7F. А в **строках 9, 10** в регистр PORTD записывается 0x7F, а в регистр DDRD — ноль. Если вы помните, те же значения мы присваивали тем же самым регистрам в программе на Ассемблере. Точнее, небольшое отличие все же есть.

В программе на Ассемблере в регистр PORTD мы записывали 0xFF (в двоичном варианте 0b11111111). В программе на СИ в тот же регистр мы записываем 0x7F (0b01111111). Эти два числа отличаются лишь значением старшего бита. Но для данного порта это несущественно, так как его старший разряд незадействован. Поэтому в каждой программе мы делаем так, как это удобнее.

После инициализации всех регистров начинается основной цикл программы (**строка 31**). **Основной цикл** — это обязательный элемент любой программы для микроконтроллера. Поэтому мастер всегда создает заготовку этого цикла. То есть создает цикл, тело которого пока не содержит не одной команды.

В том месте, где программист должен расположить команды, составляющие **тело этого цикла**, мастер помещает специальное сообщение, приглашающее вставить туда код программы. Оно гласит: Place your code here (Пожалуйста, вставьте ваш код). Мы последовали этому приглашению и вставили требуемый код (**строка 32**). В нашем случае он состоит всего из одной команды. Эта команда присваивает регистру PORTB значение регистра PORTD.

Наша программа на языке СИ готова. Выполняясь многократно в бесконечном цикле, команда присвоения постоянно переносит содержимое порта PD в порт PB, реализуя тем самым наш алгоритм.

1.3. Переключающийся светодиод

Постановка задачи

Как уже говорилось, предыдущая задача настолько проста, что решение ее средствами микропроцессорной техники лишено всякого смысла. Усложним немного задачу. **Заставим переключаться светодиод при нажатии кнопки.**

Новая задача, может звучать так:

«Разработать устройство управления одним светодиодным индикатором при помощи одной кнопки. При каждом нажатии кнопки светодиод должен поочередно включаться и отключаться. При первом нажатии кнопки светодиод должен включиться, при следующем отключиться и т. д.».

Вы можете сказать, что и эта новая задача легко решаема при помощи простейшего D-триггера. Однако все же рассмотрим, как ее можно решить при помощи микроконтроллера.

Принципиальная схема

Так как для новой задачи, как и для предыдущей, нам необходима всего одна кнопка и всего один светодиод, то придумывать новую схему не имеет смысла. Применим для второй задачи уже знакомую нам схему, показанную на **рис. 1.2**.

Алгоритм

Алгоритм задачи номер два начинается так же, как алгоритм нашей первой задачи. То есть с набора команд, выполняющих инициализацию системы. Так как схема и принцип работы портов ввода—вывода не изменились, то алгоритм инициализации системы будет полностью повторять соответствующий алгоритм из предыдущего примера.

После команд инициализации начинается основной цикл программы. Однако действия, выполняемые основным циклом, будут немного

другими. Попробуем, как и в предыдущем случае, описать эти действия словами.

1. Прочитать состояние младшего разряда порта PD (PD.0).
2. Если значение этого разряда равно единице, перейти к началу цикла.
3. Если значение разряда PD.0 равно нулю, изменить состояние выхода PB.0 на противоположное.
4. Перейти к началу цикла.

Итак, мы описали алгоритм словами. Причем это довольно общее описание. Реальный алгоритм немного сложнее. Словесное описание алгоритма не всегда удобно. Гораздо нагляднее графический способ описания алгоритма. На **рис. 1.3** алгоритм нашей работы будущей программы изображен **в графическом виде**.

Такой способ отображения информации называется **графом**. Прямоугольниками обозначаются различные действия, выполняемые программой. Суть выполняемого действия вписывается внутрь такого прямоугольника.

Допускается объединять несколько операций в один блок и обозначать одним прямоугольником. Последовательность выполнения действий показывается стрелками. Ромбик реализует разветвление программы. Он представляет собой операцию выбора. Условие выбора записывается внутри ромбика. Если условие истинно, то дальнейшее выполнение программы продолжится по пути, обозначенному словом «Да».

Если условие не выполнено, то программа пойдет по другому пути, обозначенному стрелкой с надписью «Нет». Прямоугольником со скругленными боками принято обозначать начало и конец алгоритма. В нашем случае программа не имеет конца. Основной цикл программы является бесконечным циклом.

Рассмотрим подробнее алгоритм, изображенный на **рис. 1.8**. Как видно из рисунка, сразу после старта программы выполняется установка вершины стека. Следующее действие — это программирование портов ввода—вывода. Затем начинается главный цикл программы (обведен пунктирной линией). Внутри цикла ход выполнения программы разветвляется.

Первой операцией цикла является проверка состояния младшего разряда порта PD (PD0). Программа сначала читает состояние этой линии, а затем выполняет операцию сравнения. В процессе сравнения значение разряда PD0 проверяется на равенство единице. Если

условие выполняется, программа переходит к началу цикла (по стрелке «Да»).

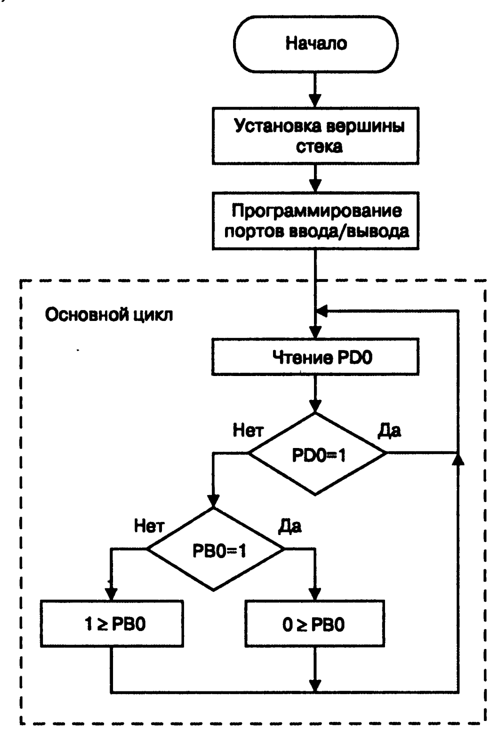


Рис. 1.8. *Алгоритм программы с переключающимся светодиодом*

Если условие не выполняется (PD0 не равен единице), выполнение программы продолжается по стрелке «Нет», где выполняется еще одна операции сравнения. Это сравнение является частью процедуры переключения светодиода. Для того, чтобы переключить светодиод, мы должны проверить его текущее состояние и перевести его в противоположное.

Как вы помните, светодиодом управляет младший разряд порта PB (PB0). Поэтому именно его мы будем проверять и изменять. **Работа алгоритма** переключения светодиода предельно проста. Сначала оператор сравнения проверяет разряд PB0 на равенство единице. Если результат проверки — истина (PB0=1), то разряд сбрасывается в ноль (0 => PB0). Если ложно, устанавливается в единицу (1 => PB0).

Сочетание символов «=>» означает **операцию присвоения**. Такое обозначение иногда используется в программировании при написании алгоритмов. После переключения светодиода управление передается на начало главного цикла.

Итак, наш алгоритм готов, и можно **приступать к составлению программы**. Но не торопитесь. Все не так просто. Приведенный выше алгоритм хорош лишь для теоретического изучения приемов программирования. На практике же он работать не будет.

Дело в том, что микроконтроллер работает с такой скоростью, что за время, пока человек будет удерживать кнопку в нажатом состоянии, главный цикл программы успеет выполниться многократно (до сотни раз). Это произойдет даже в том случае, если человек постарается нажать и отпустить кнопку очень быстро. При каждом проходе главного цикла программа обнаружит факт нажатия кнопки и переключит светодиод.

В результате работа нашего устройства будет выглядеть следующим образом. Как только кнопка будет нажата, светодиод начнет быстро переключаться. На столько быстро, что вы даже не увидите, как он мерцает. Это будет выглядеть как свечение в полнакала.

В момент отпускания кнопки процесс переключения остановится, и светодиод окажется в одном из своих состояний (засветится или потухнет). В каком именно состоянии он останется, зависит от момента отпускания кнопки. А это случайная величина. Как видите, описанный выше алгоритм не позволяет создать устройство, соответствующее нашему техническому заданию.

Для того, чтобы решить данную проблему, нам необходимо **усовершенствовать наш алгоритм**. Для этого в программу достаточно ввести **процедуру ожидания**. Процедура ожидания приостанавливает основной цикл программы сразу после того, как произойдет переключение светодиода. Теперь программа должна ожидать момента отпускания кнопки. Как только кнопка окажется отпущенной, выполнение главного цикла возобновляется.

Новый, доработанный алгоритм приведен на **рис. 1.9**. Как видно из рисунка, новый алгоритм дополнен всего двумя новыми операциями, которые и реализуют цикл ожидания. Цикл ожидания добавлен после процедуры переключения светодиода. Выполняя цикл ожидания, программа сначала читает значение бита PD0, а затем проверяет его на равенство единице. Если PD0 не равно единице (кнопка нажата), то цикл ожидания повторяется. Если PD0 равно единице (кнопка

отпущена) то цикл ожидания прерывается, и управление перейдет на начало основного цикла.

Новый алгоритм вполне работоспособен и может стать основой реальной программы. Попробуем составить такую программу.

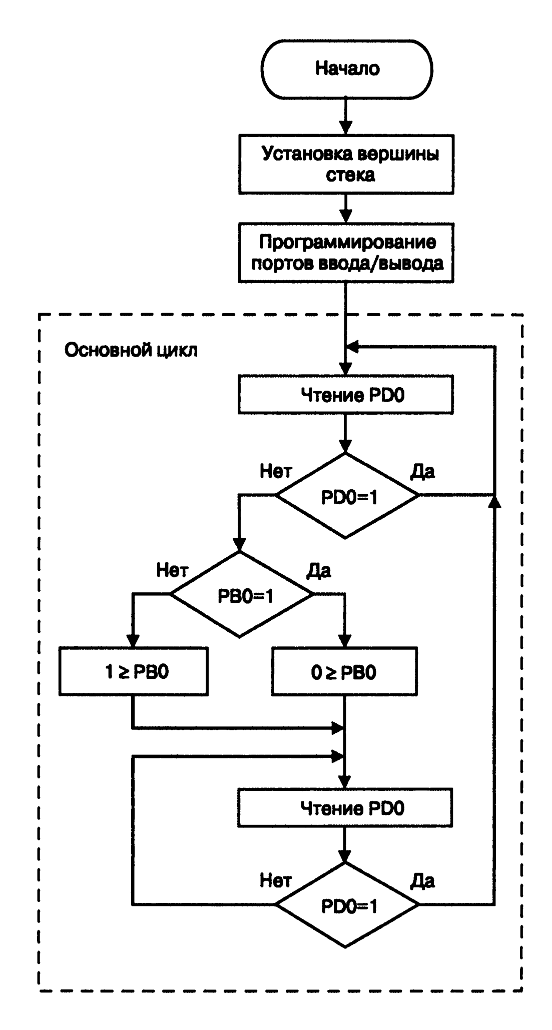


Рис. 1.9. *Усовершенствованный алгоритм программы с переключающимся светодиодом*

Программа на Ассемблере

Текст возможного варианта программы для второго примера приведен в **листинге 1.3.**

Листинг 1.3

```
;###############################################
;##               Пример 2                    ##
;##     Программа переключения светодиода     ##
;###############################################

;------------------------ Псевдокоманды управления

.include "tn2313def.inc"        ; Присоединение файла описаний
.list                           ; Включение листинга

.def temp = R16                 ; Определение главного рабочего регистра

;------------------------ Начало программного кода

         .cseg                  ; Выбор сегмента программного кода
         .org    0              ; Установка текущего адреса на ноль

;------------------------- Инициализация стека

         ldi     temp, RAMEND   ; Выбор адреса вершины стека
         out     SPL, temp      ; Запись его в регистр стека

;------------------------- Инициализация портов ВВ

         ldi     temp, 0        ; Записываем ноль в регистр temp
         out     DDRD, temp     ; Записываем этот ноль в DDRD (порт PD на ввод)

         ldi     temp, 0xFF     ; Записываем число $FF в регистр temp
         out     DDRB, temp     ; Записываем temp в DDRB (порт PB на вывод)
         out     PORTB, temp    ; Записываем temp в PORTB (потушить светодиод)
         out     PORTD, temp    ; Записываем temp в PORTD (включаем внутр.резист.)

;-------------------------- Инициализация компаратора

         ldi     temp, 0x80     ; Выключение компаратора
         out     ACSR, temp

;------------------------- Начало основного цикла

main:    in      temp, PIND     ; Читаем содержимое порта PD
         sbrc    temp, 0        ; Проверка младшего разряда
         rjmp    main           ; Если не ноль, переходим в начало

;------------------------------- Переключение светодиода

         in      temp, PINB     ; Читаем содержимое порта PB
         sbrc    temp, 0        ; Проверка младшего разряда
         rjmp    m1
         sbi     PORTB, 0       ; Установка выход PB0 в единицу
         rjmp    m2
m1:      cbi     PORTB, 0       ; Сброс PB0 в ноль

;------------------------------- Цикл ожидания отпускания кнопки

m2:      in      temp, PIND     ; Читаем содержимое порта PD
         sbrs    temp, 0        ; Проверка младшего разряда
         rjmp    m2             ; Продолжить ожидание отпускания кнопки
         rjmp    main           ; К началу цикла
```

В программе применены следующие новые для нас команды:

sbrc

Команда из группы условных переходов. Вызывает пропуск следующей за ней команды, если соответствующий разряд РОН сброшен. У команды два параметра. Первый параметр — **имя регистра общего назначения**, второй параметр — **номер проверяемого бита. В строке 17** программы (**листинг 1.3**) подобная команда проверяет нулевой разряд регистра `temp`. Если этот разряд равен нулю, то команда, записанная в **строке 16**, пропускается, и выполняется команда из **строки 17**. Если проверяемый бит равен единице, то пропуска не происходит, и выполняется команда в **строке 16**.

sbrs

Команда, обратная предыдущей. Пропускает следующую команду, если соответствующий разряд РОН установлен в единицу. Имеет те же два параметра, что и команда `sbrc`. **В строке 26 (листинг 1.3)** подобная команда проверяет значение младшего разряда регистра `temp`. Если проверяемый бит равен единице, то команда в **строке 27** пропускается, и выполняется команда в **строке** 28. Если проверяемый разряд равен нулю, то выполняется **строка 27**.

sbi

Сброс в ноль одного из разрядов порта ввода—вывода. Команда имеет два параметра: имя порта и номер сбрасываемого разряда. В **строке 22** (**листинг 1.3**) подобная команда выполняет сброс младшего разряда порта `PORTB`.

cbi

Установка в единицу одного из разрядов порта ввода—вывода. Имеет такие же два параметра, как и предыдущая команда. В **строке 24** (**листинг 1.3**) подобная команда устанавливает младший разряд порта `PORTB` в единицу.

Описание программы (листинг 1.3)

Первая часть программы (**строки 1—15**) полностью повторяет аналогичную часть программы из предыдущего примера (**листинг 1.1**). И это неудивительно, так как алгоритм инициализации не изменился. Зато значительно **усложнился основной цикл программы**. Теперь он значительно вырос по объему и занимает **строки 16—28**. **В строке 16** производится чтение порта `PORTD`. Число, прочитанное из порта, записывается в регистр `temp`.

В **строке 17** производится проверка младшего разряда прочитанного числа. Если младший бит равен единице (кнопка не нажата), то управление переходит к **строке 18**. **В строке 18** находится оператор безусловного перехода, который передает управление по метке `main`, то есть на начало цикла. Таким образом, пока кнопка не нажата. будет выполняться короткий цикл программы (**строки 16, 17** и **18**).

Если кнопка нажата, младший разряд числа в регистре `temp` окажется равным нулю. В этом случае оператор `sbrc` в **строке 17** передаст управление к **строке 19**, где начинается модуль переключения светодиода. И начинается он с чтения состояния порта PB.

В **строке 20** производится проверка младшего бита считанного числа. Если этот бит равен нулю, то **строка 21** пропускается, и выполняется **строка 22**. Если младший бит равен единице, то выполняется **строка 21**. В **строке 22** оператор `sbi` устанавливает младший бит регистра `PORTB` в единицу.

А в **строке 21** находится оператор безусловного перехода, который передает управление по метке `m1` на **строку 24**. Там оператор `cbi` сбрасывает младший бит регистра `PORTB` в ноль. Таким образом, происходит переключение в младшем разряде порта PB. Ноль меняется на единицу, а единица на ноль.

После переключения светодиода управление передается на **строку 25**. Это происходит либо при помощи команды безусловного перехода (**строка 23**), либо естественным путем после **строки 24**.

Строки 25—27 содержат **цикл ожидания момента отпускания кнопки**. Цикл ожидания начинается с чтения содержимого порта `PORTD` (**строка 25**). Прочитанное значение записывается в регистр `temp`. Затем производится проверка младшего разряда прочитанного числа (**строка 26**). Если этот разряд равен нулю (кнопка еще не отпущена), выполняется **строка 27** (безусловный переход на метку `m2`), и цикл ожидания повторяется снова.

Когда при очередной проверке кнопка окажется отпущенной, повинуясь команде sbrc (в **строке 26**), микроконтроллер пропустит **строку 27** и перейдет к **строке 28**. Расположенный там безусловный переход передаст управление на начало основного цикла (по метке `main`).

Программа на языке СИ

Та же задача на языке СИ решается следующим образом. При помощи построителя создаем точно такую же заготовку программы с теми же параметрами, как и в предыдущем случае. **Доработка** программы также будет сводиться к вписыванию необходимых команд в основной цикл программы. Однако это будут другие команды, реализующие новый алгоритм.

Возможный вариант программы смотри в **листинге 1.4**. Прежде чем мы перейдем к изучению этой программы, необходимо остановиться на новом элементе языка СИ, который в ней применяется.

if else

Условный оператор. Этот оператор позволяет выполнять разные операции в зависимости от некоторого условия. В программе на языке СИ оператор записывается следующим образом:

```
if (условие)
{ Набор операторов 1 }
else
{ Набор операторов 2 }
```

Условие — это любое логическое выражение. Если результат этого выражения — истина (не равен 0), то выполняется «Набор операторов 1». В противном случае выполняется «Набор операторов 2».

Оба набора операторов — это любые допустимые операторы языка СИ. Каждый из операторов в наборе должен оканчиваться точкой с запятой. Добавочное слово `else` не обязательно. Его можно исключить вместе с набором операторов 2. Тогда, если условие ложно, оператор не будет выполнять никаких действий.

Листинг 1.4

```
/*****************************************************
Project : Prog2
Пример 2
Управление светодиодом

Chip type           : ATtiny2313
Clock frequency     : 4,000000 MHz
Data Stack size     : 32
*****************************************************/

#include <tiny2313.h>

void main(void)
{
CLKPR=0x80;  // Отключить деление частоты системного генератора
CLKPR=0x00;

PORTA=0x00;  // Инициализация порта A
DDRA=0x00;

PORTB=0xFF;  // Инициализация порта B
DDRB=0xFF;

PORTD=0x7F;   // Инициализация порта D
DDRD=0x00;

TCCR0A=0x00;  // Инициализация таймера/счетчика 0
TCCR0B=0x00;
TCNT0=0x00;
OCR0A=0x00;
OCR0B=0x00;

TCCR1A=0x00;  // Инициализация таймера/счетчика 1
TCCR1B=0x00;
TCNT1H=0x00;
TCNT1L=0x00;
ICR1H=0x00;
ICR1L=0x00;
OCR1H=0x00;
OCR1L=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

GIMSK=0x00;   // Инициализация внешних прерываний
MCUCR=0x00;

TIMSK=0x00;   // Инициализация прерываний от таймеров

USICR=0x00;   // Инициализация универсального последовательного интерфейса

ACSR=0x80;    // Инициализация аналогового компаратора

while (1)
    {
          while (PIND.0==1) {}
          if (PINB.0==1)
             { PORTB.0=0; }
          else
             { PORTB.0=1; }
          while (PIND.0==0) {}
    };
}
```

Описание программы (листинг 1.4)

Начало программы (до **строки 30**) сформировано автоматически и полностью соответствует соответствующей части предыдущей программы (**листинг 1.2**). Я лишь немного сократил комментарии для того, чтобы не перегружать текст программы лишней информацией.

Тело основного цикла претерпело значительные изменения. Теперь он занимает **строки 31—37**. В **строке 32** расположена процедура ожидания нажатия кнопки. Она представляет собой пустой цикл `while`. В теле цикла (две фигурные скобки) нет ни одного оператора.

Цикл не выполняет никаких действий. Он будет выполняться, пока его условие истинно. В качестве условия выбрано равенство младшего разряда регистра PORTD нулю. На языке СИ это записывается следующим образом:

```
PORTD.0==1.
```

В языке СИ различают **оператор равенства** и **оператор присвоения**:

- один символ «=» означает присвоение,

  запись типа `A=5` означает присвоение переменной А значения 5;
- двойной символ «==» означает операцию сравнения,

  запись `A==5` означает проверку на равенство значений переменной `A` и константы `5`.

Результат такого сравнения равен единице в случае, если `A` равно пяти, и равен нулю, если это не так. Поэтому, цикл `while` в **строке 32** программы продолжается до тех пор, пока значение разряда PORTD.0 равно единице. То есть до тех пор, пока кнопка, подключенная к этому разряду, остается не нажатой. Как только кнопка окажется нажатой, цикл (**строка 32**) заканчивается, и программа перейдет к **строке 33**.

В строках 33—36 находится **оператор сравнения**. Он выполняет **задачу переключения светодиода**. Для этого в его условии записана проверка младшего разряда порта PB, то есть содержимого регистра PINB. Разряд проверяется на равенство единице (**строка 33**).

Если значение разряда равно единице, то выполняется **строка 34**, в которой младшему разряду регистра PORTB присваивается нулевое значение. Если условие не выполняется (значение PINB.0 не равно единице), выполняется **строка 36,** и младшему разряду PORTB присваивается единица. Значение, записанное в регистр PORTB, непосредственно поступает на выход порта PB. В результате состояние младшего разряда порта (PB0) меняется на противоположное.

В **строке 37** программы расположен **цикл ожидания отпускания кнопки**. Он аналогичен циклу в **строке 32**. Только условие теперь обратное. Цикл выполняется до тех пор, пока значение PORTD.0 равно нулю. То есть пока кнопка нажата.

1.4. Боремся с дребезгом контактов

Постановка задачи

Обратимся еще раз к схеме на **рис. 1.2**. В схеме используется кнопка, имеющая одну группу из двух нормально разомкнутых контактов. А если есть контакты, значит, есть и дребезг этих контактов. В [3] рассматривается способ борьбы с антидребезгом аппаратным способом. Теперь рассмотрим **способ борьбы с дребезгом контактов программным путем.**

Итак, новая задача будет сформулирована следующим образом:

«Разработать схему управления светодиодом при помощи одной кнопки. При нажатии кнопки светодиод должен изменять свое состояние на противоположное (включен или выключен). При разработке программы принять меры для борьбы с дребезгом контактов».

Схема

Как уже говорилось, принципиальная схема остается прежняя (см. **рис. 1.2**).

Алгоритм

Алгоритм нам придется доработать. Самый простой способ борьбы с дребезгом — введение в программу специальных задержек. Рассмотрим это подробнее. Начнем с исходного состояния, когда контакты кнопки разомкнуты. Программа ожидает их замыкания. В момент замыкания возникает дребезг контактов.

Дребезг приводит к тому, что на соответствующем разряде порта PD вместо простого перехода с единицы в ноль мы получим серию

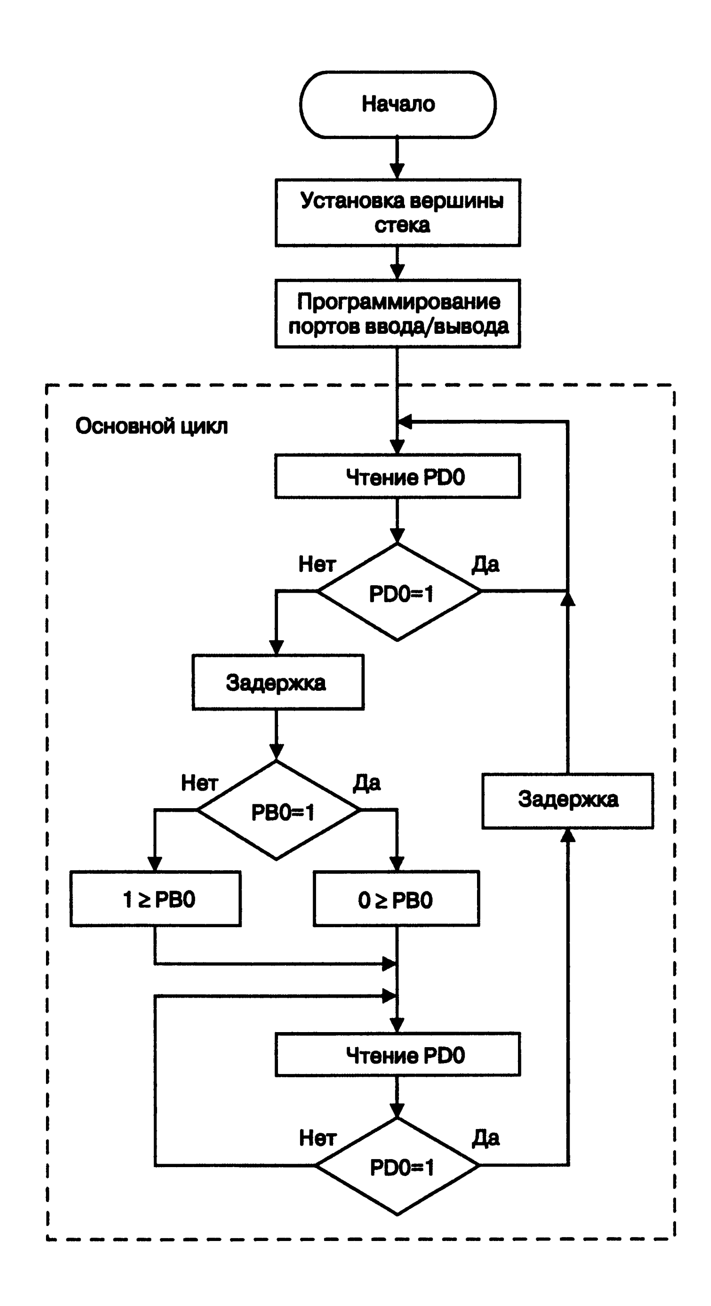


Рис. 1.10. *Алгоритм управления светодиодом с антидребезгом*

импульсов. Для того, чтобы избавится от их паразитного влияния, программа должна сработать следующим образом. Обнаружив первый же нулевой уровень на входе, программа должна перейти в режим ожидания. В режиме ожидания программа приостанавливает все свои действия и просто отрабатывает задержку.

Время задержки должно быть выбрано таким образом, чтобы оно превышало время дребезга контактов. Такую же процедуру задержки нужно ввести в том месте программы, где она ожидает отпускания кнопки. Для разработки нового алгоритма возьмем за основу предыдущий (см. **рис. 1.9**). Доработанный алгоритм с добавлением операций антидребезговой задержки приведен на **рис. 1.10**. Как вы можете видеть из рисунка, вся доработка свелась к включению двух процедур задержки. Одной — после обнаружения факта нажатия кнопки, а второй — после обнаружения факта ее отпускания.

Программа на Ассемблере

Для реализации нового алгоритма немного доработаем программу (**листинг 1.3**). Новый вариант программы приведен ниже (**листинг 1.5**). В этой программе используются следующие новые для нас операторы:

rcall

Переход к подпрограмме. У этого оператора всего один параметр — относительный адрес начала подпрограммы. Относительный адрес — это просто смещение относительно текущего адреса. Выполняя команду `rcall`, микроконтроллер запоминает в стеке текущий адрес программы из счетчика команд и переходит по адресу, определяемому смещением. Такой же принцип задания адреса для перехода мы уже встречали в команде `rjmp`. В **строке 20** программы (**листинг 1.5**) производится вызов подпрограммы задержки по адресу, соответствующему метке `wait`.

ret

Команда выхода из подпрограммы. По этой команде микроконтроллер извлекает из стека адрес, записанный туда при выполнении команды `rcall`, и осуществляет передачу управления по этому адресу. В **листинге 1.5** команду `ret` мы можем видеть в конце подпрограммы `wait` (см. **строку 37**).

push

Запись содержимого регистра общего назначения в стек. У данного оператора всего один операнд — имя регистра, содержимое которого нужно поместить в стек. В **строке 32** программы (**листинг 1.5**) в стек помещается содержимое регистра с именем `loop`.

pop

Извлечение информации из стека. У этого оператора тоже всего один операнд — имя регистра, в который помещается информация, извлекаемая из стека. В **строке 36** программы (**листинг 1.5**) информация извлекается из стека и помещается в регистр `loop`.

dec

Уменьшение содержимого РОН. У команды один параметр — имя регистра. Команда `dec` (декремент) уменьшает на единицу содержимое регистра, имя которого является ее параметром. В **строке 34** программы (**листинг 1.5**) уменьшается на единицу содержимое регистра `loop`.

brne

Оператор условного перехода (переход по условию). У этого оператора всего один параметр — относительный адрес перехода. Условие перехода звучит как «не равно». Попробуем разобраться, как проверяется это условие. И тут нам придется вспомнить о регистре состояния микроконтроллера (SREG).

Как уже говорилось ранее, каждый бит этого регистра представляет собой флаг. Все флаги регистра предназначены для управления работой микроконтроллера. Кроме уже известного нам флага I (глобальное разрешение прерываний), этот регистр имеет ряд флагов, отражающих результаты работы различных операций.

Полное описание регистра SREG смотрите [4]. В данном случае нас интересует лишь один из таких флагов — **флаг нулевого результата (флаг Z)**. Этот флаг устанавливается в том случае, когда при выполнения очередной команды результат окажется равным нулю. **Например** при вычитании двух чисел, сдвиге разрядов числа или в результате операции сравнения. В нашем случае на значение флага влияет команда `dec`. (**строка 34**). Если в результате действия этого

оператора содержимое регистра окажется равным нулю, то флаг Z устанавливается в единицу. В противном случае он сбрасывается в ноль.

Флаг Z будет хранить записанное в него значение до тех пор, пока какая-нибудь другая команда его не изменит. Какие из команд оказывают влияние на флаг Z, а какие нет, можно узнать из приложения.

Команда `brne` использует флаг Z в качестве условия. Команда выполняет переход только в том случае, если флаг Z сброшен. То есть когда результат предыдущей команды неравен нулю. В нашей программе (**листинг 1.5**) подобный оператор применяется в **строке 35**.

Листинг 1.5

```
    ;###############################################
    ;##               Пример 3                    ##
    ;##     Программа переключения светодиода     ##
    ;##       с использованием антидребезга       ##
    ;###############################################

    ;------------------------- Псевдокоманды управления

1   .include "tn2313def.inc"          ; Присоединение файла описаний
2   .list                             ; Включение листинга

3   .def temp = R16                   ; Определение главного рабочего регистра
4   .def loop = R17                   ; Определение регистра организации цикла

    ;------------------------- Начало программного кода

5           .cseg                     ; Выбор сегмента программного кода
6           .org    0                 ; Установка текущего адреса на ноль

    ;-------------------------- Инициализация стека

7           ldi     temp, RAMEND      ; Выбор адреса вершины стека
8           out     SPL, temp         ; Запись его в регистр стека

    ;-------------------------- Инициализация портов ВВ

9           ldi     temp, 0           ; Записываем ноль в регистр temp
10          out     DDRD, temp        ; Записываем этот ноль в DDRD (порт PD на ввод)

11          ldi     temp, 0xFF        ; Записываем число $FF в регистр temp
12          out     DDRB, temp        ; Записываем temp в DDRB (порт PB на вывод)
13          out     PORTB, temp       ; Записываем temp в PORTB (потушить светодиод)
14          out     PORTD, temp       ; Записываем temp в PORTD (включаем внутр.резист.)

    ;--------------------------- Инициализация компаратора

15          ldi     temp, 0x80        ; Выключение компаратора
16          out     ACSR, temp

    ;------------------------- Начало основного цикла

17  main:   in      temp, PIND        ; Читаем содержимое порта PD
18          sbrc    temp, 0           ; Проверка младшего разряда
19          rjmp    main              ; Если не ноль, переходим в начало
20          rcall   wait              ; Вызов подпрограммы задержки
```

```
   ;------------------------------ Переключение светодиода

21         in      temp, PINB      ; Читаем содержимое порта PB
22         sbrc    temp, 0         ; Проверка младшего разряда
23         rjmp    m1
24         sbi     PORTB, 0        ; Установка выход PB0 в единицу
25         rjmp    m2
26   m1:   cbi     PORTB, 0        ; Сброс PB0 в ноль

   ;------------------------------ Цикл ожидания отпускания кнопки

27   m2:   in      temp, PIND      ; Читаем содержимое порта PD
28         sbrs    temp, 0         ; Проверка младшего разряда
29         rjmp    m2              ; Продолжить ожидание отпускания кнопки

30         rcall   wait            ; Вызов подпрограммы задержки
31         rjmp    main            ; К началу цикла

   ;-------------------------------- Подпрограмма задержки

32   wait: push    loop            ; Сохраняем содержимое регистра loop
33         ldi     loop, 200       ; Помещаем в loop константу задержки
                                   ; Цикл задержки
34   wt1:  dec     loop            ; Уменьшаем значение регистра loop
35         brne    wt1             ; Если не ноль, продолжаем цикл

36         pop     loop            ; Восстанавливаем значение регистра loop
37         ret                     ; Выход из подпрограммы
```

Описание программы (листинг 1.5)

Новый вариант программы является полной копией старой (см. **листинг 1.3**), в которую добавлены новые элементы, **обеспечивающие антидребезговую задержку**. Так как задержка нужна в двух разных местах программы, она оформлена в виде подпрограммы. Для формирования задержки используется один **дополнительный регистр общего назначения.**

Поэтому в начале нашей новой программы (**строка 4**) добавлена **команда описания регистра.** Регистру `r17` и присваивается имя `loop`. По-английски слово `loop` означает **цикл.** Таким именем принято называть переменные, применяемые для организации циклов. Не удивляйтесь, что я употребил тут термин «переменная». В языке Ассемблер тоже используется понятие «переменные». Так наш регистр `loop` можно считать переменной с именем `loop`.

Запись значения в этот регистр эквивалентна присвоению значения переменной. Также можно интерпретировать и другие операции с регистром. Сложение содержимого двух регистров можно считать сложением переменных, вычитание — вычитанием, и так далее.

Подпрограмма задержки расположена в **строках 32—37**. Первой **строке** подпрограммы присвоена метка `wait`. Именно по этой метке и будет вызываться подпрограмма. Опустим пока назначение команд `hush` и `pop` (**строки 32 и 36**). Собственно процедура задержки рас-

положена в **строках 33—35**. Формирование задержки производится путем многократного выполнения пустого цикла. Сначала в регистр `loop` записывается некое начальное значение (**строка** 33). В нашем случае оно равно 200.

Затем начинается цикл, который постепенно уменьшает значение регистра `loop` до нуля (**строки 34 и 35**). Происходит это следующим образом. В **строке 34** содержимое регистра уменьшается на единицу, а в **строке 35** происходит проверка содержимого на ноль. Если ноль не достигнут, то управление передается по метке `wt1`, и цикл повторяется. Когда же содержимое `loop` кажется равным нулю, очередного перехода не произойдет, и цикл задержки закончится.

Очевидно, что в нашем случае цикл задержки выполнится 200 раз. Если обратиться к приложению, то можно узнать, что команда `dec` выполняется за один такт системного генератора. Команда `brne` выполняется:

- за один такт, если не вызывает перехода;
- за два такта, если вызывает переход.

Поэтому один цикл задержки будет выполняться за 3 такта. Двести циклов за 3×200=600 тактов. Тактовая частота кварцевого резонатора у нас равна 4 МГц. Длительность одного колебания тактовой частоты равна 1/4 = 0,25 мкс. Поэтому время, за которое будут выполнены все 200 циклов задержки, равно 600×0,25=150 мкс. Добавьте сюда время выполнения остальных команд подпрограммы, включая команду вызова подпрограммы и команду возврата из подпрограммы, и вы получите общее время задержки (162 мкс).

Максимальная задержка, которую можно сформировать при помощи данной подпрограммы, равна (255×3×0,25)+12=203,25 мкс. Учтите, что в нашем случае не применяется предварительное деление частоты тактового генератора. Если это было бы не так, то длительность выполнения каждой команды нужно было бы умножать на коэффициент деления предварительного делителя.

Теперь вернемся **к двум командам работы со стеком**, которые мы не стали рассматривать вначале. Они предназначены для сохранения в стеке и последующего восстановления содержимого регистра `loop`. В начале подпрограммы (**строке 32**) значение `loop` сохраняется, а перед выходом их подпрограммы (**строка 36**) — восстанавливается.

Подобный прием придает программе одно **полезное свойство**. После окончания работы подпрограммы значения всех регистров микроконтроллера остается без изменений. В данном конкретном случае

такое свойство ничего не дает, кроме, разве что, **дополнительной задержки**. Однако в сложных программах, имеющих не одну, а несколько подпрограмм, одни и те же регистры удобно использовать в разных подпрограммах.

Те же самые регистры может использовать и основная программа. В этом случае описанное выше полезное свойство просто необходимо для правильной работы всей программы. Зная эту особенность, программисты стараются применять подобный прием в каждой подпрограмме, независимо от того, полезен он в данном конкретном случае или нет.

Не исключена ситуация, когда в процессе доработки программы вам все же придется повторно использовать какие-либо регистры. Заранее обеспечить корректную работу вашей подпрограммы — это хороший стиль программирования.

В соответствии с алгоритмом (**рис. 1.5**) подпрограмма задержки в нашей программе вызывается два раза. **Первый раз** — после окончания цикла ожидания нажатия кнопки (**строка 20**). **Второй раз** — после окончания цикла ожидания отпускания (**строка 30**).

Программа на языке СИ

С программой на языке СИ мы поступим так же, как с программой на Ассемблере. Мы просто возьмем предыдущий вариант (**листинг 1.4**) и вставим в него *задержки*. Для языка СИ добавить задержку в программу гораздо проще, чем для Ассемблера. Для того, чтобы ввести задержку, мы воспользуемся стандартной библиотекой процедур задержки.

Эта библиотека входит в состав **программного комплекса CodeVisionAVR**. Название этой библиотеки `delay.h`. Посмотрите на новый вариант программы (**листинг 1.6**). В **строке 2** мы присоединяем библиотеку `delay.h` к тексту нашей программы. Так же, как в **строке 1** мы присоединили файл описания микросхемы. После присоединения библиотеки в нашем распоряжении появляется несколько функций, реализующих задержку. Воспользуемся одной из них. Имя этой функции `delay_us` (задержка в микросекундах). Она обеспечивает задержку в любое целое количество микросекунд.

Количество микросекунд задержки передается в функцию в качестве параметра. В **строке 34** программы (**листинг 1.6**) функция задержки вызывается

первый раз. Она обеспечивает задержку на 200 мкс после окончания цикла ожидания нажатия кнопки. В **строке 40** такая же задержка вызывается после окончания цикла ожидания отпускания кнопки.

Теперь немного поговорим **о функции `delay_us`**. Данная функция обеспечивает формирование задержки при помощи бесконечного цикла. Такого же цикла, который мы применяли в программе на

Листинг 1.6

```
/*****************************************************
Project : Prog3
Пример 3
Управление светодиодом

Chip type           : ATtiny2313
Clock frequency     : 4,000000 MHz
*****************************************************/

#include <tiny2313.h>
#include <delay.h>

void main(void)
{
PORTB=0xFF;  // Инициализация порта B
DDRB=0xFF;

PORTD=0x7F;   // Инициализация порта D
DDRD=0x00;

ACSR=0x80;   // Инициализация аналогового компаратора

while (1)
     {
           while (PIND.0==1) {}
           delay_us(200);
           if (PINB.0==1)
              { PORTB.0=0; }
           else
              { PORTB.0=1; }
           while (PIND.0==0) {}
           delay_us(200);
     };
}
```

Ассемблере. Но теперь нам не нужно описывать цикл в подробностях. Достаточно применить готовую функцию.

В процессе трансляции пустой цикл формируется автоматически. Начальное значение высчитывается, исходя из заданной величины задержки и частоты тактового генератора, указанной при создании проекта.

Кроме новой для нас функции, программа, показанная на **листинге 1.6**, имеет еще несколько отличий от оригинала:

- в программе существенно сокращен блок команд инициализации;
- удалены все команды, созданные построителем, которые дублируют запись в регистры значений по умолчанию.

Удаление лишних команд сокращает объем программы и облегчает ее понимание. Какие же команды были удалены? Это команды настройки тех систем, которые в данном случае не используются. Оставлены лишь команды настройки портов В и D. А также команда настройки аналогового компаратора.

1.5. Мигающий светодиод

Постановка задачи

Создадим **программу с мигающим светодиодом**. Сформулируем условие следующим образом:

«Создать устройство с одним светодиодом и одной управляющей кнопкой. Кнопка должна включать и выключать мигание светодиода. Пока кнопка отпущена, светодиод не должен светиться. Все время, пока кнопка нажата, светодиод должен мигать с частотой 5 Гц».

Схема

И в этом примере мы воспользуемся уже знакомой нам схемой, изображенной на **рис. 1.2**.

Алгоритм программы

Алгоритм такой программы тоже состоит из **алгоритма начальной установки** и **алгоритма основного цикла**. Начальная установка в данном случае не отличается от начальной установки всех предыдущих примеров. Алгоритм основного цикла программы можно описать следующим образом:

1. Произвести чтение порта PD;
2. Проверить младший разряд полученного числа (если его значение равно нулю, включить алгоритм мигания);
3. Если значение младшего разряда PD равно единице, выключить алгоритм мигания и потушить светодиод;
4. Перейти к началу основного цикла (пункт1).

Для того, чтобы выполнить все предыдущие пункты, нам нужно описать **алгоритм мигания светодиода**. Он будет выглядеть следующим образом:

5. Зажечь светодиод;
6. Выдержать паузу;
7. Потушить светодиод;
8. Выдержать паузу;
9. Перейти к началу алгоритма мигания (пункт 1).

Программа на Ассемблере

Возможный вариант программы приведен в **листинге 1.7**.

Листинг 1.7

```
;###############################################
;##                Пример 4                  ##
;##            Мигающий светодиод            ##
;###############################################

;------------------------- Псевдокоманды управления

.include "tn2313def.inc"         ; Присоединение файла описаний
.list                            ; Включение листинга
.def temp = R16                  ; Определение главного рабочего регистра
.def loop1 = R17                 ; Определение первого регистра организации цикла
.def loop2 = R18                 ; Определение второго регистра организации цикла
.def loop3 = R19                 ; Определение третьего регистра организации цикла

;------------------------- Начало программного кода

        .cseg                    ; Выбор сегмента программного кода
        .org     0               ; Установка текущего адреса на ноль

;------------------------- Инициализация стека

        ldi      temp, RAMEND       , Выбор адреса вершины стека
        out      SPL, temp          ; Запись его в регистр стека

;------------------------- Инициализация портов ВВ

        ldi      temp, 0         ; Записываем ноль в регистр temp
        out      DDRD, temp      ; Записываем этот ноль в DDRD (порт PD на ввод)

        ldi      temp, 0xFF      ; Записываем число $FF в регистр temp
        out      DDRB, temp      ; Записываем temp в DDRB (порт PB на вывод)
        out      PORTB, temp     ; Записываем temp в PORTB (потушить светодиод)
        out      PORTD, temp     ; Записываем temp в PORTD (включаем внутр.резист.)

;--------------------------- Инициализация компаратора

        ldi      temp, 0x80      ; Выключение компаратора
        out      ACSR, temp

;------------------------- Начало основного цикла

main:   sbi      PORTB, 0        ; Устанавливаем PB0 в единицу (тушим светодиод)
        in       temp, PIND      ; Читаем содержимое порта PD
        sbrc     temp, 0         ; Проверка младшего разряда
        rjmp     main            ; Если не ноль, переходим в начало

;-------------------------------- Мигание светодиода

        cbi      PORTB, 0        ; Устанавливаем PB0 в единицу (тушим светодиод)
        rcall    wait1           ; Вызов подпрограммы задержки

        sbi      PORTB, 0        ; Сброс PB0 в ноль (зажигаем светодиод)
        rcall    wait1           ; Вызов подпрограммы задержки

        rjmp     main            ; К началу цикла

;--------------------------------- Подпрограмма задержки

        push     loop1           ; Сохраняем содержимое регистра loop1
        push     loop2           ; Сохраняем содержимое регистра loop2
        push     loop3           ; Сохраняем содержимое регистра loop3
```

```
31          ldi     loop3, 15    ; Помещаем в loop3 константу задержки
32  wt1:    dec     loop3        ; Уменьшаем значение регистра loop3
33          breq    wt4
34          ldi     loop2, 100   ; Помещаем в loop2 константу задержки
35  wt2:    dec     loop2        ; Уменьшаем значение регистра loop2
36          breq    wt1
37          ldi     loop1, 255   ; Помещаем в loop1 константу задержки
38  wt3:    dec     loop1        ; Уменьшаем значение регистра loop1
39          brne    wt3          ; Если не ноль, продолжаем цикл
40          rjmp    wt2

41  wt4:    pop     loop3        ; Восстанавливаем значение регистра loop3
42          pop     loop2        ; Восстанавливаем значение регистра loop2
43          pop     loop1        ; Восстанавливаем значение регистра loop1
44          ret                  ; Выход из подпрограммы
```

Программа содержит всего одну новую для нас команду.

breq

Оператор условного перехода по условию «равно». Этот оператор — полная противоположность оператору `brne`, описанному в предыдущем примере. Отличие этих двух операторов друг от друга в том, что `brne` вызывает переход в том случае, если флаг Z установлен, а оператор `breq`, напротив, вызовет переход, если Z сброшен.

Описание программы (листинг 1.7)

Для новой задачи нам пришлось создать **новую подпрограмму задержки**. Это произошло потому, что приведенная в предыдущем примере подпрограмма не способна обеспечить задержку достаточно большой длительности. Новая подпрограмма задержки использует не один, а целых три вложенных друг в друга цикла. По этой причине нам понадобится не один, а три вспомогательных регистра.

Поэтому в блок инициализации новой программы включены три оператора, определяющие три вспомогательные переменные `loop1`, `loop2` и `loop3` (**строки 4, 5, 6**). В остальном блок инициализации полностью соответствует аналогичному блоку из предыдущего примера. Теперь он занимает **строки 1—18**.

Основной цикл программы занимает **строки 19—27**. Он начинается с установки единицы в младшем разряде порта PB (**строка 19**). В результате светодиод выключается. Следующая команда читает содержимое порта PD и помещает его в регистр `temp` (**строка 20**).

В **строке 21** содержимое младшего разряда полученного числа проверяется на равенство единице. Если младший разряд равен единице (кнопка отпущена), то управление передается по метке `main`. И цикл

замыкается. Так происходит все время, пока кнопка не нажата. При каждом проходе оператор `sbi` (**строка 19**) подтверждает единицу на выходе PB0. Светодиод остается незажженным.

Как только кнопка будет нажата, младший бит считанного из порта PD числа окажется равным нулю. Повинуясь команде сравнения в **строке 21**, микроконтроллер пропустит **строку 22**, и управление перейдет к **строке 23**. В **строке 23** начнется процедура мигания светодиода.

Она реализует один цикл мигания и работает следующим образом. Сначала оператор `cbi` (**строка 23**) устанавливает на выходе PB0 низкий логический уровень (зажигает светодиод). Затем происходит вызов подпрограммы задержки (**строка 24**). По окончании задержки команда `sbi` (**строка 25**) переводит разряд PB0 в единицу (тушит светодиод). И снова задержка (**строка 26**).

Оператор безусловного перехода (**строка 27**) передает управление на начало основного цикла программы. И вся процедура повторится сначала. Снова проверка нажатия кнопки. Если кнопка нажата, то цикл мигания повторяется. Если же кнопка окажется отпущенной, продолжения мигания не произойдет. Программа потушит светодиод и войдет в цикл ожидания нажатия кнопки (**строки 19—22**).

И так с миганием мы разобрались. Теперь **перейдем к подпрограмме формирования задержки.** Текст этой подпрограммы занимает **строки 28—44**. Так как требуемая частота мигания должна быть равна 5 Гц, подпрограмма должна обеспечивать время задержку 1/5 = 0,2 с (200 мс).

Как уже говорилось, подпрограмма представляет собой три **вложенных друг в друга цикла**. Самый внутренний цикл организован при помощи регистра `loop1` и включает в себя **строки 38 и 39**. Перед началом цикла в регистр `loop1` записывается число 255 (**строка 37**). Поэтому цикл повторяется 255 раз. Число 255 — это самое большое значение, которое можно записать в один восьмиразрядный регистр. Как уже говорилось, задержка, формируемая таким циклом, может быть лишь чуть больше, чем 200 мкс.

Для увеличения задержки организован **второй цикл**. Второй цикл использует регистр `loop2`. Перед началом цикла в этот регистр записывается число 100 (**строка 34**). Цикл организован при помощи оператора `dec` (**строка 35**), который последовательно уменьшает содержимое регистра `loop2` оператора сравнения `breq` (**строка 36**), проверяет, не достигло ли значение регистра нуля.

Пока в `loop2` не равно нулю, выполняется тело цикла (**строки 37—40**). В тело второго цикла включен первый цикл, использующий регистр

`loop1`. Таким образом, цикл `loop1` выполняется при каждом проходе цикла `loop2`. Общее суммарное количество проходов обоих циклов будет равно 255×100 = 25500.

Но и этого недостаточно для создания нужной задержки. Даже если начальное значение для `loop2` мы выберем равным 255, и тогда мы не получим искомые 200 мс. Поэтому вокруг первых двух циклов организован третий. **Третий цикл** использует регистр `loop3` и построен точно так же, как второй. Перед началом работы в регистр `loop3` записывается число 15 (**строка 31**). Выполнение цикла обеспечивают оператор `dec` (**строка 32**) и оператор сравнения (**строка 33**). При каждом проходе цикла `loop3` выполняются вложенные циклы `loop1` и `loop2`. В результате общее количество проходов строенного цикла возрастает еще в 15 раз, что обеспечивает требуемую задержку.

Кроме построенного цикла, подпрограмма задержки содержит уже знакомые нам операторы сохранения и восстановления используемых регистров. В нашем случае подпрограмма использует три регистра. Поэтому в начале подпрограммы содержимое всех трех регистров сохраняется в стеке (**строки 28, 29, 30**).

Перед выходом из подпрограммы содержимое всех этих регистров восстанавливается (**строки 41, 42, 43**). Обратите внимание, что восстановление регистров происходит в порядке, обратном порядку их запоминания. Регистр, который был записан в стек последним, извлекается первым.

Программа на языке СИ

Как вы уже наверно догадываетесь, осуществить задержку в программе, написанной на языке СИ, будет гораздо проще, чем на Ассемблере. **Листинг 1.8** содержит один из вариантов подобной программы. Программа не содержит новых для нас операторов, поэтому сразу перейдем к ее описанию. Для создания задержки используется та же самая библиотека подпрограмм, что и **в предыдущем примере** (**листинг 1.6**). Однако, в нашем случае, мы берем другую функцию из этой библиотеки. Функцию `delay_ms` (задержка в миллисекундах).

Рассмотрим подробнее **работу программы**. Все команды инициализации в новой программе взяты из предыдущего примера и перенесены оттуда без изменений. **Различия** начинаются в **главном цикле программы**.

Листинг 1.8

```
        /*************************************************
        Project : Prog4
        Пример 4
        Мигающий светодиод

        Chip type          : ATtiny2313
        Clock frequency    : 4,000000 MHz
        *************************************************/

1       #include <tiny2313.h>
2       #include <delay.h>

3       void main(void)
        {
4       PORTB=0xFF;    // Инициализация порта B
5       DDRB=0xFF;

6       PORTD=0x7F;    // Инициализация порта D
7       DDRD=0x00;

8       ACSR=0x80;     // Инициализация аналогового компаратора

9       while (1)
            {
10              if (PIND.0==1)      // Проверка нажатия кнопки
11                 { PORTB.0=1; }  // Тушим светодиод
12              else
                   {
13                    PORTB.0=1;       // Тушим светодиод
14                    delay_ms(200);   // Задержка
15                    PORTB.0=0;       // Зажигаем светодиод
16                    delay_ms(200);   // Задержка
                   }
            };
        }
```

Оператор `if` в **строке** 10 производит проверку младшего разряда регистра PD на равенство единице. Если разряд равен единице (кнопка не нажата), то выполняется **строка 11** (запись в PB0 единицы). Эта **строка** выполняется все время, пока кнопка не нажата. В этом случае светодиод остается потушенным. Если нажать кнопку, младший разряд PD окажется равным нулю. В этом случае вместо **строки 11** выполняются **строки 13—16.**

Они представляют собой **процедуру мигания светодиода.** Эта процедура работает следующим образом. В **строке 13** светодиод тушится. Затем осуществляется задержка на 200 мс (**срока 14**). В **строке 15** светодиод зажигается. После этого опять осуществляется задержка (**строка 16**). То есть **выполняется один цикл мигания.**

Так как вся конструкция if—else находится внутри основного цикла, после окончания цикла мигания все операции повторяются сначала. То есть снова выполняется проверка состояния кнопки (**строка 10**), а по результатам проверки — одно их вышеописанных действий. В случае, если кнопка все еще нажата, цикл мигания повторяется. Если кнопка отпущена — просто гасится светодиод.

1.6. Бегущие огни

Постановка задачи

В прежние времена очень популярны среди радиолюбителей были различные автоматы световых эффектов. Сейчас этим не удивить, и совсем недорого можно купить готовую мигающую световую гирлянду. Однако, как пример для программирования, такая задача вполне подойдет. Итак, **разрабатываем «Бегущие огни».**

Задание будет звучать следующим образом:

«Разработать автомат «Бегущие огни» для управления составной гирляндой из восьми отдельных гирлянд. Устройство должно обеспечивать «движения» огня в двух разных направлениях. Переключение направления «движения» должно осуществляться при помощи переключателя».

Схема

В соответствии с поставленной задачей наше устройство должно управлять восемью световыми гирляндами. Удобно задействовать для этого все восемь выходов одного из портов. Кроме того, нам придется подключать переключатель направления. Для этого нам понадобится еще один порт. Очевидно, что для такой задачи вполне подойдет уже знакомый нам микроконтроллер ATtiny2313.

Для создания и отладки программы совсем не обязательно подключать к микроконтроллеру гирлянды лампочек. Для начала подключим просто восемь светодиодов. Для подключения настоящей гирлянды каждый светодиод нужно заменить ключевой схемой на тиристоре, к которой уже подключить гирлянду. Примеры ключевых схем легко найти в радиолюбительской литературе. Схема бегущих огней со светодиодами приведена на **рис. 1.11**.

Как видно из **рис. 1.11**, схема представляет собой доработанный вариант схемы управления светодиодом (см. **рис. 1.2**). К предыдущей схеме просто добавлены еще семь дополнительных светодиодов, включенных таким же образом, как и светодиод VD1.

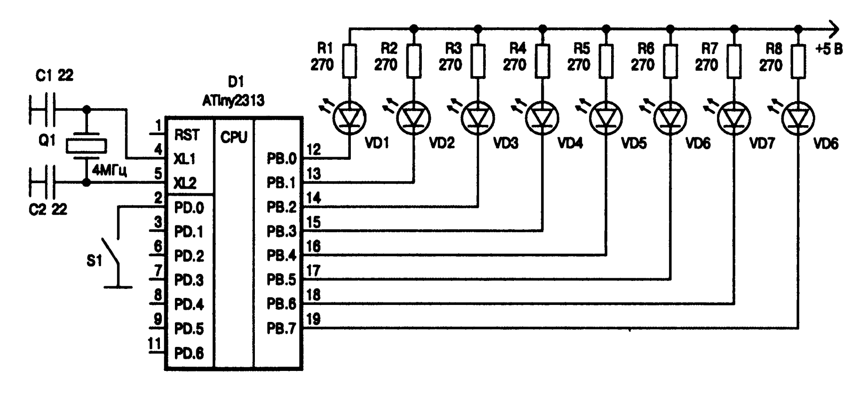


Рис. 1.11. *Схема автомата «Бегущие огни»*

Алгоритм

Для создания эффекта «бегущих огней» удобнее всего воспользоваться операторами сдвига, которые имеются в системе команд любого микроконтроллера. Такие операторы сдвигают содержимое одного из регистров микроконтроллера на один разряд влево или вправо. Если сдвигать содержимое регистра и после каждого сдвига выводить новое содержимое в порт PB, подключенные к нему светодиоды будут загораться поочередно, имитируя бегущий огонь. Алгоритм работы бегущих огней может быть разный. Один из возможных алгоритмов в самых общих чертах будет выглядеть следующим образом:

1. Считать состояние переключателя управления;
2. Если контакты переключателя разомкнуты, перейти к процедуре сдвига вправо;
3. Если контакты замкнуты, перейти к процедуре сдвига влево;
4. После окончания полного цикла сдвига (восемь последовательных сдвигов) перейти к началу алгоритма, то есть к пункту 1.

Таким образом, все время, пока контакты переключателя разомкнуты, программа будет выполнять сдвиг вправо. Если состояние переключателя не изменилось, сдвиг в прежнем направлении продолжается. Если замкнуть контакты переключателя, то все время, пока они замкнуты, будет выполняться сдвиг влево. Как при сдвиге вправо, так и при сдвиге влево после каждого полного цикла сдвига (8 шагов)

происходит проверка переключателя. Если его состояние не такое же, как и прежде, то направление сдвига не изменяется. В противном случае программа меняет направление сдвига.

Выполнение алгоритма сдвига

Посмотрим теперь, как выполняется сам алгоритм сдвига. Сдвиг влево и сдвиг вправо выполняются аналогично. Ниже приводится обобщенный алгоритм для сдвига влево и сдвига вправо, снабженный комментариями.

1. Записать в рабочий регистр начальное значение. В качестве начального значения используется двоичное число, у которого один из разрядов равен единице, а остальные разряды равны нулю. Для сдвига вправо нам нужно число с единицей в самом старшем разряде (0b10000000). Для сдвига влево в единицу устанавливается младший разряд (0b00000001).
2. Вывести значение рабочего регистра в порт PB.
3. Вызвать подпрограмму задержки. Задержка нужна для того, чтобы скорость «бега» огней была нормальная для глаз наблюдателя. Если бы не было задержки, то скорость «бега» была бы столь велика, что мы бы и не увидели движения огней. С точки зрения наблюдателя мерцание огней выглядело бы как слабое свечение всех светодиодов.
4. Сдвинуть содержимое рабочего регистра вправо (влево) на один разряд.
5. Проверить, не окончился ли полный цикл сдвига (8 шагов).
6. Если полный цикл сдвига не закончен, перейти к пункту 2 данного алгоритма. Это приведет к тому, что пункты 2, 3, 4, 5 и 6 повторятся 8 раз, и лишь затем завершится полный цикл сдвига.

Программа на Ассемблере

Возможный вариант программы приведен ниже (см. **листинг 1.9**). В программе встречается несколько новых операторов. Кроме того, мы будем иметь дело с новым для нас флагом. Этот флаг также является одним из разрядов регистра SREG и называется **флагом переноса**.

Определение. ***Флаг переноса** — это разряд, куда помещается бит переноса при выполнении операций сложения двух чисел или бит заема при операциях вычитания.*

Содержимое флага переноса так же, как и содержимое флага нулевого результата Z, может служить условием для оператора условного перехода. Кроме своего основного предназначения, флаг переноса иногда выполняет и вспомогательные функции. **Например**, он участвует во всех операциях сдвига в качестве дополнительного разряда. Теперь рассмотрим подробнее все новые операторы.

lsr

Логический сдвиг вправо. Этот оператор имеет всего один параметр — имя регистра, содержимое которого сдвигается. Схематически данная операция выглядит следующим образом:

$$0 \rightarrow d7 \rightarrow d6 \rightarrow d5 \rightarrow d4 \rightarrow d3 \rightarrow d2 \rightarrow d1 \rightarrow d0 \rightarrow C.$$

То есть содержимое младшего разряда переносится в флаг переноса C, на его место поступает содержимое разряда 1, в разряд 1 попадает содержимое разряда 2, и так далее. В самый старший разряд записывается ноль.

lsl

Логический сдвиг влево. Действие этого оператора обратно действию предыдущего. Схема такого сдвига выглядит следующим образом:

$$C \leftarrow d7 \leftarrow d6 \leftarrow d5 \leftarrow d4 \leftarrow d3 \leftarrow d2 \leftarrow d1 \leftarrow d0 \leftarrow 0.$$

То есть в данном случае в C попадает содержимое старшего разряда. Содержимое всех остальных разрядов сдвигается на один шаг влево. В самый младший разряд записывается ноль.

brcc

Переход по условию «нет переноса». Данный оператор проверяет содержимое флага переноса C и осуществляет переход по относительному адресу в том случае, если флаг C не установлен (равен нулю).

eor

Оператор «Исключающее ИЛИ». Этот оператор имеет два параметра. В качестве параметров выступают имена двух регистров, с содержимым которых производится данная операция. Оператор производит поразрядную операцию «Исключающее ИЛИ» между содержимым обоих регистров. Результат помещается в тот регистр, имя которого указано в качестве первого параметра.

Листинг 1.9

```
;################################################
;##                Пример 5                    ##
;##              "Бегущие огни"                ##
;################################################

;------------------------- Псевдокоманды управления

.include "tn2313def.inc"          ; Присоединение файла описаний
.list                             ; Включение листинга

.def  temp = R16                  ; Определение главного рабочего регистра
.def  loop1 = R17                 ; Определение первого регистра организации цикла
.def  loop2 = R18                 ; Определение второго регистра организации цикла
.def  loop3 = R19                 ; Определение третьего регистра организации цикла
.def  rab = R20                   ; Определение рабочего регистра для команд сдвига

;------------------------- Начало программного кода

        .cseg                     ; Выбор сегмента программного кода
        .org    0                 ; Установка текущего адреса на ноль

;-------------------------- Инициализация стека

        ldi     temp, RAMEND      ; Выбор адреса вершины стека
        out     SPL, temp         ; Запись его в регистр стека

;-------------------------- Инициализация портов ВВ

        ldi     temp, 0           ; Записываем ноль в регистр temp
        out     DDRD, temp        ; Записываем этот ноль в DDRD (порт PD на ввод)

        ldi     temp, 0xFF        ; Записываем число $FF в регистр temp
        out     DDRB, temp        ; Записываем temp в DDRB (порт PB на вывод)
        out     PORTB, temp       ; Записываем temp в PORTB (потушить светодиод)
        out     PORTD, temp       ; Записываем temp в PORTD (включаем внутр.резист.)

;--------------------------- Инициализация компаратора

        ldi     temp, 0x80        ; Выключение компаратора
        out     ACSR, temp

;-------------------------- Начало основного цикла
```

```
main:   in      temp, PIND          ; Читаем содержимое порта PD
        sbrc    temp, 0             ; Проверка младшего разряда
        rjmp    m3                  ; Если не ноль, переходим в начало

;------------------------- Сдвиг вправо

m1:     ldi     rab, 0b10000000     ; Запись начального значения
m2:     ldi     temp, 0xFF
        eor     temp, rab           ; Инверсия содержимого регистра rab
        out     PORTB, rab          ; Вывод текущего значения в порт PB
        rcall   wait1               ; Задержка
        lsr     rab                 ; Сдвиг содержимого рабочего регистра
        brcc    m2                  ; Если не дошло до конца регистра продолжить
        rjmp    main                ; На начало

;------------------------- Сдвиг влево

m3:     ldi     rab, 0b00000001     ; Запись начального значения
m4:     ldi     temp, 0xFF
        eor     temp, rab           ; Инверсия содержимого регистра rab
        out     PORTB, rab          ; Вывод текущего значения в порт PB
        rcall   wait1               ; Задержка
        lsl     rab                 ; Сдвиг содержимого рабочего регистра
        brcc    m4                  ; Если не дошло до конца регистра продолжить
        rjmp    main                ; На начало

;-------------------------------- Подпрограмма задержки

        push    loop1               ; Сохраняем содержимое регистра loop1
        push    loop2               ; Сохраняем содержимое регистра loop2
        push    loop3               ; Сохраняем содержимое регистра loop3

        ldi     loop3, 15           ; Помещаем в loop3 константу задержки
wt1:    dec     loop3               ; Уменьшаем значение регистра loop3
        breq    wt4
        ldi     loop2, 100          ; Помещаем в loop2 константу задержки
wt2:    dec     loop2               ; Уменьшаем значение регистра loop2
        breq    wt1
        ldi     loop1, 255          ; Помещаем в loop1 константу задержки
wt3:    dec     loop1               ; Уменьшаем значение регистра loop1
        brne    wt3                 ; Если не ноль, продолжаем цикл
        rjmp    wt2

wt4:    pop     loop3               ; Восстанавливаем значение регистра loop3
        pop     loop2               ; Восстанавливаем значение регистра loop2
        pop     loop1               ; Восстанавливаем значение регистра loop1
        ret                         ; Выход из подпрограммы
```

Описание программы (листинг 1.9)

Как уже говорилось ранее, модуль инициализации новой программы остался таким же, как в предыдущих примерах. В новой программе он занимает **строки 1—19**. Дополнен лишь блок описания переменных. Кроме уже знакомых нам регистров `loop1`, `loop2` и `loop3`, нам понадобится еще один дополнительный регистр. Этот регистр мы будем использовать как рабочий в операциях сдвига. В **строке 7** в качестве такого регистра выбран регистр `r20`, которому присваивается имя `rab`.

В **строке 20** начинается **основной цикл программы**. И начинается он с чтения содержимого порта PD. Результат помещается в регистр `temp`. В **строке 21** происходит оценка младшего разряда прочитанного числа. Если этот разряд равен единице, то оператор безусловного перехода в **строке 22** пропускается, и программа переходит к выполнению процедуры сдвига вправо (начало процедуры — **строка 23**). Если младший разряд считанного из PD числа равен нулю, то оператор `rjmp` в **строке 22** передает управление по метке m3, и программа переходит к процедуре сдвига влево (начало процедуры — **строка 31**).

Процедура «сдвиг вправо» работает следующим образом. В **строке 23** рабочему регистру `rab` присваивается начальное значение. Для наглядности это число записано в двоичном формате. Затем начинается **цикл сдвига (строки 24—30)**. Первой операцией цикла сдвига, в соответствии с алгоритмом, должна быть операция вывода содержимого регистра `rab` в порт PB. Однако существует одно небольшое препятствие.

Если просто вывести содержимое `rab` в порт PB, то мы получим картину, обратную той, которая нам необходима. Все светодиоды, кроме одного, будут светиться. Это произойдет потому, что ноль на выходе порта зажигает светодиод, а единица тушит. Если мы хотим получить бегущий огонь, а не бегущую тень, нам нужно проинвертировать содержимое регистра `rab` перед тем, как вывести в порт PB.

Для инвертирования содержимого регистра `rab` воспользуемся командой `eor` («Исключающее ИЛИ»). Операция «Исключающее ИЛИ» обладает способностью инвертирования чисел. Если вы вернетесь назад и посмотрите на таблицу истинности операции «Исключающее ИЛИ» (**рис. 1.8**), то вы можете заметить эту особенность.

Правило. *Для всех строк таблицы истинности справедливо правило: если один из операндов равен единице, то результат операции равен инверсному значению второго операнда.*

Поэтому, если произвести операцию «Исключающее ИЛИ» между двумя байтами, значение одного из которых будет равно 0xFF, то в результате этой операции мы получим инверсное значение второго байта. Для выполнения такой операции используется вспомогательный регистр `temp`. В **строке 24** в регистр `temp` записывается число 0xFF. В **строке 25** производится операция «Исключающее ИЛИ» между содержимым регистров `temp` и `rab`.

Результат этой операции помещается в `temp`, так как именно он является первым параметром данной команды. Содержимое регистра `rab` при этом не изменяется. В **строке 26** содержимое регистра `temp` выводится в порт PB.

Следующий этап процедуры сдвига — **вызов подпрограммы задержки**. Вызов этой подпрограммы происходит в **строке 27**. В **строке 28** производится сдвиг содержимого регистра `rab` на один бит вправо. В **строке 29** оператор `brcc` проверяет состояние признака переноса. Эта проверка позволяет обнаружить момент, когда закончится полный цикл сдвига. Как это происходит, иллюстрирует **табл. 1.2**.

Таблица 1.2. *Сдвиг информации в рабочем регистре*

Шаг	b7	b6	b5	b4	b3	b2	b1	b0	C
1	1	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0
5	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	1

В таблице показаны значения всех разрядов вспомогательного регистра `rab` для каждого из восьми шагов, составляющих **полный цикл сдвига**. Разряды сдвигаемого регистра обозначены как b7—b0. Последний столбец показывает содержимое признака переноса. Как видно из таблицы, единица, которая в начале находится в самом старшем разряде, при каждом очередном шаге сдвигается в соседнюю позицию.

В результате, после восьмого шага она оказывается в ячейке признака переноса. Пока С равно нулю, оператор `brcc` в **строке 29** передает управление по метке m2, и цикл сдвига продолжается. После восьмого шага признак переноса С станет равен единице. Поэтому перехода

на начало цикла в **строке 29** не произойдет, и управление перейдет к **строке 30**. В результате очередного девятого цикла сдвига не произойдет. Оператор безусловного перехода в **строке 30** передаст управление на начало основного цикла, и программа снова приступит к проверке состояния кнопки.

Процедура сдвига влево занимает **строки 31—38**. Эта процедура работает точно так же, как и процедура сдвига вправо. **Отличия:**

- начальное значение, записываемое в регистр `rab` (см. **строку 31**), равно 0b00000001;
- вместо оператора `lsr` (сдвиг вправо) в **строке 36** использован оператор `lsl` (сдвиг влево).

В качестве подпрограммы задержки применена уже известная нам **подпрограмма с тремя вложенными циклами**. Текст этой подпрограммы полностью скопирован из предыдущего примера (**листинг 1.7**) и расположен в **строках 39—55**.

Программа на языке СИ

Возможный вариант той же программы, но на языке СИ, приведен в **листинге 1.10**. В этой программе **впервые** мы будем использовать **переменную**. До сих пор мы не применяли переменные лишь благодаря предельной простоте предыдущих программ. Теперь же переменная нам понадобится для того, чтобы осуществлять операции сдвига.

Переменная будет хранить текущее значение всех сдвигаемых битов так же, как в программе на Ассемблере их хранил регистр `rab`. Назовем переменную тем же именем, каким мы называли регистр. Описание переменной в нашей программе происходит в **строке 4**. Так как сдвигаемых битов должно быть всего восемь, то самый подходящий тип данных для нашей переменной — это `unsigned char`.

Переменная такого типа имеет длину в один байт. Второй однобайтовый тип (`char`) в данном случае нам не подходит, так как представляет собой число со знаком. У такого числа старший разряд интерпретируется как знак. И лишь семь младших разрядов используются непосредственно для хранения значений.

Строки 1—9 составляет **модуль инициализации программы**. В полном соответствии с алгоритмом модуль инициализации новой программы почти полностью повторяет соответствующий модуль из предыдущего примера. **Исключение** составляет лишь вновь добавленная **строка** с описанием переменной (**строка 4**).

Листинг 1.10

```
   /*****************************************************
   Project : Prog5
   Пример 5
   Бегущие огни

   Chip type            : ATtiny2313
   Clock frequency      : 4.000000 MHz
   Data Stack size      : 32
   *****************************************************/

1  #include <tiny2313.h>
2  #include <delay.h>

3  void main(void)
   {
4  unsigned char rab;  // Вводим переменную rab

5  PORTB=0xFF;  // Инициализация порта B
6  DDRB=0xFF;

7  PORTD=0x7F;   // Инициализация порта D
8  DDRD=0x00;

9  ACSR=0x80;   // Инициализация аналогового компаратора

10 while (1)
       {
11     if (PIND.0==1)  // Проверка состояния переключателя
            {                    // Сдвиг вправо
12          rab = 0b10000000;    // Запись начального значения
13          while (rab!=0)
                 {
14                    PORTB=rab^0xFF;  // Запись в порт с инверсией
15                    rab = rab >> 1;  // Сдвиг разрядов
16                    delay_ms (200);  // Задержка в 200 мсек
                 }
            }
17     else
            {                    // Сдвиг влево
18          rab = 0b00000001;    // Запись начального значения
19          while (rab!=0)
                 {
20                    PORTB=rab^0xFF;  // Запись в порт с инверсией
21                    rab = rab << 1;  // Сдвиг разрядов
22                    delay_ms (200);  // Задержка в 200 мсек
                 }
            }
       };
   }
```

Основной цикл программы занимает **строки 10—22**. **Телом цикла** является **оператор сравнения** (конструкция `if—else`), проверяющий состояние бита, связанного с переключателем. Собственно проверка происходит в **строке 11**. Здесь младший бит порта PD проверяется на равенство единице. Если он равен единице, то выполняется процедура сдвига вправо (**строки 12—16**). В противном случае выполняется процедура сдвига влево (**строки 18—22**).

Каждая из этих процедур выполняет цикл из восьми сдвигов в нужном направлении. Так как вся конструкция `if—else` находится внутри бесконечного цикла, то она многократно повторяется. То есть после

проверки происходит восемь сдвигов в нужном направлении. Затем новая проверка, и так далее.

Обе процедуры сдвига построены одинаково. Рассмотрим подробнее **процедуру сдвига вправо.** В **строке 12** переменной rab присваивается начальное значение. Затем начинается цикл сдвига. **Цикл организован** при помощи оператора while (**строка 13**). Его тело составляют **строки 14—16**, которые реализуют уже знакомый нам алгоритм. Сначала происходит вывод значения всех разрядов переменной rab в порт PB. Как и в предыдущем случае, выводимое значение нам нужно предварительно проинвертировать.

Для инвертирования числа мы снова используем прием, который мы применили в программе на Ассемблере. То есть, воспользуемся оператором «Исключающее ИЛИ». Обратимся к **строке 14** нашей программы. В этой **строке** регистру PORTB присваивается значение выражения rab^0xFF. Символ «^» в языке СИ как раз и означает операцию «Исключающее ИЛИ». При помощи единственного выражения мы сразу и инвертируем и присваиваем.

В **строке 15** производится **сдвиг разрядов.** Для сдвига используется оператор «>>». Результатом выражения rab >> 1 является число, полученное путем сдвига всех разрядов переменной rab на одну позицию вправо. Число 1 справа от оператора сдвига означает количество разрядов, на которое нужно сдвинуть число. Таким образом, выражение

```
rab = rab >> 1;
```

означает: присвоить переменной rab значение этой же переменной сдвинутое на один разряд вправо. Язык СИ допускает другую, сокращенную форму записи того же самого выражения:

```
rab >>= 1;
```

Новое выражение полностью эквивалентно предыдущему. Подобные изящные сокращения являются фирменной особенностью языка СИ. Благодаря ним программа на языке СИ получается короче и проще. В **строке 16** вызывается функция задержки. Время задержки составляет 200 мс.

Для того, чтобы цикл сдвига повторялся только восемь раз, используется оператор цикла (**строка 13**). В качестве условия, при котором цикл выполняется, используется выражение rab !=0. В языке СИ выражение «!=» означает «Не равно».

Таким образом, наш цикл сдвига (**строки 14—16**) будет выполняться до тех пор, пока rab не равен нулю. Это и будут наши восемь шагов

сдвига. Чтобы убедиться в этом, еще раз посмотрите на **табл. 1.2**. Значение одного из разрядов b0—b7, а, значит, и всей переменной `rab`, во время первых восьми шагов не равно нулю. И только на девятом шаге все восемь рабочих разрядов обнулятся. Но так как при этом заложенное нами условие не выполняется, последнего девятого цикла не будет.

Процедура сдвига влево находится в **строках 18—22** программы и работает точно так же, как процедура сдвига вправо. Имеются лишь два отличия:

- другое начальное значение переменной `rab` (см. **строку 18**);
- применен другой оператор сдвига.

Для сдвига влево применяется оператор «<<» (см. **строку 21**). При желании выражение в **строке 21** тоже можно сократить. Вместо `rab = rab << 1;` можно записать `rab <<= 1;`

1.7. Использование таймера

Постановка задачи

В предыдущих примерах для формирования задержки мы использовали один или несколько вложенных программных циклов. Однако такой способ приемлем далеко не всегда. **Главный недостаток** подобного метода состоит в том, что он полностью загружает центральный процессор. Пока микроконтроллер занят формированием задержки, он не может выполнять никаких других задач.

Еще один недостаток — невозможно с достаточной точностью выбрать время задержки. Гораздо лучшие результаты дает другой способ — **формирование интервалов времени при помощи одного из встроенных таймеров/счетчиков микроконтроллера**. Любой из таймеров/счетчиков может работать как с использованием прерываний, так и без прерываний. Далее мы рассмотрим оба эти варианта. И начнем мы с более простого случая.

Итак, заново сформулируем нашу задачу:

"Доработать программу «Бегущие огни», изменив процедуру формирования задержки. Новая процедура должна использовать один из внутренних таймеров/счетчиков и не использовать прерывания".

Схема

Так как мы разрабатываем не самостоятельное устройство, а лишь усовершенствуем управляющую программу, то схема устройства не изменяется.

Алгоритм

Как известно, в микроконтроллере ATtiny2313 имеются два встроенных таймера-счетчика. Поэтому сначала нам нужно выбрать, какой из них мы будем использовать. Исходить будем из заданного времени задержки 200 мс. Как известно, для формирования временных интервалов таймер/счетчик просто подсчитывает тактовые импульсы от системного генератора.

Частота сигнала этого генератора в нашем случае равна 4 МГц. А период импульсов 1/4=0,25 мкс. Для того, чтобы получить на выходе 200 мс, необходимо иметь коэффициент деления, равный $200 \cdot 10^{-3}/0{,}25 \cdot 10^{-6} = 800 \cdot 10^{3}$ (восемьсот тысяч раз).

Микросхема ATiny2313 содержит два таймера. Один восьмиразрядный и один шестнадцати. **Восьмиразрядный таймер** имеет максимальный коэффициент пересчета $2^8=256$, а шестнадцатиразрядный — $2^{16}=65536$. То есть даже шестнадцатиразрядного таймера нам не хватит для формирования требуемой задержки. Придется воспользоваться предварительным делителем. Этот делитель производит предварительное деление тактового сигнала перед тем, как тот поступит на вход таймера/счетчика.

Программным путем можно выбрать один из четырех фиксированных коэффициентов деления (см. **приложение**). Выберем самый большой возможный коэффициент деления предделителя (1024). Тогда на его выходе мы получим сигнал с частотой $4 \cdot 10^6/1024 = 3906$ Гц. Период такого сигнала будет равен $1/3906 \approx 0{,}256 \cdot 10^{-3}$ с или 0,256 мс. Именно этот сигнал поступает на вход нашего таймера, который обеспечивает окончательное деление. Посчитаем коэффициент деления, который наш таймер должен нам обеспечить: $200/0{,}256 \approx 780$. Такой коэффициент пересчета нам может обеспечить только таймер T1.

Итак, мы определились как с выбором таймера, так и с его настройками. Теперь можно приступать к созданию **новой подпрограммы задержки**. Прежде, чем это сделать, попробуем описать **алгоритм ее работы**. Данный алгоритм предполагает, что все необходимые настройки таймера предварительного делителя произведены до первого вызова подпрограммы, таймер запущен и находится в режиме непрерывного счета.

Алгоритм подпрограммы задержки представлен ниже.

1. Записать в счетный регистр таймера T1 нулевое значение.
2. Начать цикл проверки содержимого счетного регистра. В теле цикла программа должна многократно считывать содержимое счетного регистра таймера и проверять, не достигло ли оно своего конечного значения (то есть значения 780).
3. При достижении счетным регистром конечного значения, завершить цикл проверки.
4. Выйти из подпрограммы задержки.

Программа на Ассемблере

Программа «Бегущие огни» с новым вариантом подпрограммы задержки приведена в **листинге 1.11**. Новая подпрограмма задержки использует таймер T1 и описанный выше алгоритм работы. Рассмотрим подробнее, как работает такая программа. А начнем, как обычно, с описания новых для нас операторов.

.equ

Псевдооператор присвоения. Название оператора происходит от английского слова «эквивалентно» (equality). Используется для присвоения имен различным константам. В **строке 5** листинга 1.11 числу 120 присваивается имя `kdel`. Теперь в любом месте программы вместо числа 780 можно применять константу `kdel`. Имя для константы имеет то же значение, что и имя для переменной. Во-первых, по осмысленному имени легко понять назначение константы. **Например**, `kdel` расшифровывается, как «коэффициент деления». А, во-вторых, это удобно при смене значения. Поменяйте в **строке 5** число 780, **к примеру**, на 800, и везде, где бы ни встретилась константа `kdel`, она уже будет иметь новое значение.

cpi

Сравнение содержимого РОН с константой. Эта команда имеет два параметра. Первый параметр — имя регистра общего назначения, содержимое которого подлежит сравнению. Второй параметр — некая константа, с которой сравнивается содержимое РОН. По результатам сравнения устанавливаются все флаги регистра `SREG`. Флаги устанавливаются точно так же, как если бы содержимое РОН вычиталось из константы. А именно: флаг переноса C устанавливается в том случае, если при вычитании данных чисел возникает перенос в старший разряд (содержимое регистра меньше константы), и сбрасывается, если нет переноса. Флаг Z (нулевой результат) устанавливается при равенстве содержимого РОН и константы и сбрасывается в случае их неравенства. Все остальные флаги устанавливаются в соответствии со своим назначением. Подробнее об этом вы можете узнать из **приложения 1**. После того, как значения флагов определены, они могут быть использованы различными условными операторами. В **строке 44** программы (**листинг 1.11**) оператор `cpi` производит сравнение содержимого регистра `temp` с числом 0xD0.

brlo

Переход по условию «меньше». Имеется в виду, что в предыдущей команде, в результате сравнения (или вычитания) двух операндов, первый операнд оказался меньше, чем второй. Для проверки этого условия оператор использует флаг переноса С. Переход происходит лишь в том случае, если С = 1. В **строке 45** программы условный переход используется для того, чтобы передать управление на строку с меткой `wt1` в том случае, если по результатам предыдущего сравнения (**строка 44**) оказалось, что содержимое регистра `temp` меньше, чем число 0xD0.

Листинг 1.11

```
   ;###############################################
   ;##                 Пример 6                  ##
   ;##               "Бегущие огни"              ##
   ;##          с использованием таймера         ##
   ;###############################################

   ;------------------------ Псевдокоманды управления

1  .include "tn2313def.inc"         ; Присоединение файла описаний
2  .list                            ; Включение листинга

3  .def  temp = R16                 ; Определение главного рабочего регистра
4  .def  rab = R20                  ; Определение рабочего регистра для команд сдвига

5  .equ  kdel = 780

   ;------------------------ Начало программного кода

6          .cseg                    ; Выбор сегмента программного кода
7          .org       0             ; Установка текущего адреса на ноль

   ;------------------------- Инициализация стека

8          ldi        temp, RAMEND  ; Выбор адреса вершины стека
9          out        SPL, temp     ; Запись его в регистр стека

   ;------------------------- Инициализация портов ВВ

12         ldi        temp, 0       ; Записываем ноль в регистр temp
11         out        DDRD, temp    ; Записываем этот ноль в DDRD (порт PD на ввод)

12         ldi        temp, 0xFF    ; Записываем число $FF в регистр temp
13         out        DDRB, temp    ; Записываем temp в DDRB (порт PB на вывод)
14         out        PORTB, temp   ; Записываем temp в PORTB (потушить светодиод)
15         out        PORTD, temp   ; Записываем temp в PORTD (включаем внутр.резист.)

   ;------------------------- Инициализация таймера T1

16         ldi        temp, 0x05    ; Код конфигурации записываем в temp
17         out        TCCR1B, temp  ; Переносим его в регистр конфигурации таймера

   ;-------------------------- Инициализация компаратора

18         ldi        temp, 0x80    ; Выключение компаратора
19         out        ACSR, temp

   ;------------------------- Начало основного цикла

20 main:   in         temp, PIND    ; Читаем содержимое порта PD
21         sbrc       temp, 0       ; Проверка младшего разряда
```

```
22              rjmp      m3                 ; Если не ноль, переходим в начало

    ;------------------------ Сдвиг вправо

23  m1:         ldi       rab, 0b10000000    ; Запись начального значения
24  m2:         ldi       temp, 0xFF
25              eor       temp, rab          ; Инверсия содержимого регистра rab
26              out       PORTB, rab         ; Вывод текущего значения в порт PB
27              rcall     wait1              ; Задержка
28              lsr       rab                ; Сдвиг содержимого рабочего регистра
29              brcc      m2                 ; Если не дошло до конца регистра продолжить
30              rjmp      main               ; На начало

    ;------------------------ Сдвиг влево

31  m3:         ldi       rab, 0b00000001    ; Запись начального значения
32  m4:         ldi       temp, 0xFF
33              eor       temp, rab          ; Инверсия содержимого регистра rab
34              out       PORTB, rab         ; Вывод текущего значения в порт PB
35              rcall     wait1              ; Задержка
36              lsl       rab                ; Сдвиг содержимого рабочего регистра
37              brcc      m4                 ; Если не дошло до конца регистра продолжить
38              rjmp      main               ; На начало

    ;-------------------------------- Подпрограмма задержки

39  wait1:      push      temp                  ; Сохраняем содержимое регистра temp

40              ldi       temp, 0               ; Помещаем temp ноль
41              out       TCNT1H, temp          ; Записываем этот ноль в старший регистр таймера
42              out       TCNT1L, temp          ; Записываем этот ноль в младший регистр таймера

43  wt1:        in        temp, TCNT1L          ; Чтение младшей части счетного регистра
44              cpi       temp, low(kdel)       ; Сравнение с числом $0C
45              brlo      wt1                   ; Переход, если temp меньше чем kdel
46              in        temp, TCNT1H          ; Чтение старшей части счетного регистра
47              cpi       temp, high(kdel)      ; Сравнение с числом $03
48              brlo      wt1                   ; Переход, если temp меньше чем $03

49              pop       temp                  ; Восстанавливаем значение регистра temp
50              ret                             ; Выход из подпрограммы
```

Описание программы (листинг 1.11)

Как уже говорилось, данная программа является модификацией программы из предыдущего примера (см. **листинг 1.9**). **Основное отличие** новой программы от старой — полная переработка подпрограммы задержки. В связи с тем, что новая подпрограмма задержки использует таймер, для ее нормальной работы пришлось также доработать модуль инициализации основной программы.

Первая доработка модуля инициализации — команда в **строке 5**. Эта команда описывает константу `kdel`, то есть коэффициент деления таймера. Обратите внимание, что значение этой константы равно 780. Если перевести это значение в двоичную форму, то количество разрядов

такого числа будет больше восьми. А это значит, что для представления константы в двоичном виде потребуется не менее двух байтов.

Далее в программе с этой константой сравнивается содержимое счетного регистра таймера T1, который тоже имеет шестнадцать разрядов. Однако микроконтроллеры AVR работают лишь с восьмиразрядными величинами. Счетный регистр таймера T1 представляет собой два восьмиразрядных регистра TCNT1L и TCNT1H. Используемый для сравнения оператор `brlo` также работает с восьмиразрядными величинами. Как же выполняется такое сравнение?

Сравнение происходит в два этапа. Сначала сравниваются младшие разряды обеих величин, затем старшие. Младшее и старшее значение счетного регистра хранятся в двух соответствующих регистрах TCNT1L и TCNT1H. А для выделения младшего и старшего байта константы в языке Ассемблер существуют специальные функции `low` и `high`.

Рассмотрим действие этих функций на конкретном примере. Значение нашей константы `kdel` равно 780. В шестнадцатиричном виде это выглядит как 0x030C. Используя вышеописанные функции, мы можем найти старший и младший байты числа:

```
high(kdel) = 0x03 low(kdel) = 0x0C.
```

Данные функции используются в **строках** 44 и 47 программы.

Следующая доработка модуля инициализации — это две команды, выбирающие режим работы таймера (**строки 16, 17**). Эти команды записывают в регистр TCCR1B константу 0x05. В качестве вспомогательного регистра используется `temp`.

Регистр TCCR1B — это один из двух регистров выбора режимов работы таймера T1. При записи кода 0x05 в этот регистр устанавливается коэффициент предварительного деления 1/1024, и таймер переходит в режим счета. Второй регистр конфигурации таймера называется TCCR1A. Его значение нужно оставить по умолчанию. Подробнее о регистрах и режимах работы таймера смотрите в **главе 6**.

Последние изменения основной части программы коснулись команд вызова подпрограммы задержки. Вызов задержки происходит в **строках 27** и **35**. Других изменений основной части программы не потребовалось.

Новая подпрограмма задержки занимает **строки 39—50**. Начинается подпрограмма традиционно сохранением содержимого всех используемых ею регистров. В данном случае потребовалось сохранить лишь

содержимое одного регистра `temp` (см. **строку 39**). Следующие три команды производят запись нулевого значения в счетный регистр таймера T1. Сначала ноль записывается в регистр `temp` (**строка 40**). А затем содержимое `temp` поочередно помещается в регистры TCNT1H и TCNT1L (**строки 41, 42**).

Порядок записи информации в пару регистров TCNT1H, TCNT1L неслучайный. Эти два регистра обладают **свойством** так называемой **двойной буферизации**. Правила работы с такими регистрами требуют, чтобы при записи значения в эти регистры сначала записывался старший регистр TCNT1H, а потом младший TCNT1L. Дело в том, что при записи старшего байта в регистр TCNT1H он не попадает сразу по назначению, а сохраняется в специальном внутреннем регистре. Когда же поступает команда записи младшего байта в регистр TCNT1L, оба байта записываются одновременно. В этом и состоит двойная буферизация. Использование двойной буферизации позволяет менять значение счетного регистра на ходу, не останавливая таймера.

После записи нулевого значения в счетный регистр начинается **цикл проверки**. Он занимает **строки 43—48** программы. Сравнение происходит в **два этапа**. В **строках 43—45** сравнивается младшая часть счетного регистра с младшим байтом коэффициента деления. В **строках 46—48** сравниваются старшие байты. Рассмотрим это подробнее.

В **строке 43** содержимое регистра TCNT1L помещается в регистр `temp`. В **строке 44** происходит сравнение содержимого регистра `temp` с младшим байтом константы. Команда условного перехода в **строке 45** передает управление на начало цикла сравнения только в том случае, если содержимое регистра еще не достигло требуемого значения.

В **строках 46—48** такие же операции сравнения производится для регистра TCNT1H. При этом используется старший байт константы `kdel`. Если старший разряд счетного регистра не достиг требуемого значения, то управление передается на метку `wt1`. То есть в этом случае программа опять повторяет сравнение младших разрядов.

Такой порядок также диктуется наличием двойной буферизации. При чтении младшей части регистра старшая его часть запоминается в специальном внутреннем буфере. Команда чтения старшей части регистра на самом деле читает содержимое этого буфера.

Пока происходит цикл сравнения, счетчик находится в режиме счета. Содержимое счетного регистра постепенно увеличивается и, в конце концов, достигает требуемого значения. Пока происходит очередной цикл проверки, содержимое счетного регистра может даже превысить значение константы.

В этом случае переходов в **строке 45** и в **строке 48** не произойдет. В результате подпрограмма перейдет к своему завершению. В **строке 49** происходит восстановление содержимого регистра `temp`. А в **строке 50** — выход из подпрограммы.

Программа на языке СИ

Возможный вариант той же программы на языке СИ приведен в **листинге 1.12**. Эта программа представляет собой доработанный вариант программы, из предыдущего примера (**листинг 1.10**). **Причем доработка свелась к созданию новой функции задержки**. Новая функция понадобилась нам потому, что использованная ранее библиотечная функция `delay_ms` в данном случае нам не подходит.

Для формирования задержки программа использует вложенные циклы. Готовой функции, удовлетворяющей нашим новым условиям, ни в одной из стандартных библиотек не существует. Поэтому нам пришлось создать ее самостоятельно. Текст новой функции задержки приводится в **строках 2—4**. Наша **новая функция задержки** получила имя `wait1`. **Обратите внимание**, что описание функции `wait1` расположено в нашей программе раньше, чем описание функции `main`. Такой порядок отнюдь не случаен.

Правило. *В языке СИ действует правило: любая функция должна быть прежде описана и лишь затем в первый раз применена.*

Вывод. *Так как функция `main` использует функцию `wait1` в качестве процедуры задержки, то описание `wait1` должно располагаться перед описанием `main`.*

В связи с вводом новой функции нам пришлось **немного доработать и основную программу. Во-первых**, в модуль инициализации добавлены команды инициализации таймера (**строки 11, 12**). Причем команда в **строке 11** является избыточной. Нулевое значение в регистр TCCR1A можно и не записывать, так как там и так ноль по умолчанию.

Второе изменение внесено в основной цикл программы. Оно очевидно. Вместо функции задержки `delay_ms` мы применим нашу новую функцию `wait1` (см. **строки 20 и 26**). Функция `wait1` не имеет параметров, так как предназначена для формирования фиксированного значения задержки.

С этим связана **и последняя доработка**. Так как библиотечная функция задержки нам больше не нужна, мы можем исключить из программы команду, присоединяющую библиотеку `delay.h`. Других доработок основная программа не потребовала.

Теперь разберем подробнее саму функцию `wait1`. Она **формирует задержку с использованием таймера/счетчика**. Подобный алгоритм мы уже реализовывали в подпрограмме `wait1` на Ассемблере (см. **листинг 1.11**).

Однако язык **СИ** значительно **упрощает задачу**. Во-первых, нам не обязательно работать с отдельными байтами. Теперь мы без труда можем оперировать шестнадцатиразрядными числами. В результате функция задержки предельно упрощается. Она занимает всего три **строки** (**строки 2—4**). Первая **строка** — это заголовок описания функции. Из него видно, что функция `wait1` не использует параметров и не возвращает никаких величин. Тело функции составляют **строки 3 и 4**. В **строке 3** счетному регистру таймера T1 присваивается нулевое значение. Значение присваивается сразу всему шестнадцатиразрядному регистру TCNT1.

И неважно, что на самом деле микроконтроллер не имеет прямого доступа к этому регистру. После трансляции будет создана программа в машинных кодах, которая запишет сначала старшую часть (TCNT1H), а затем младшую часть (TCNT1L) регистра, строго соблюдая правила работы с регистрами, имеющими двойную буферизацию.

В **строке 4** расположен **цикл проверки**. Это пустой цикл, в качестве условия которого выступает выражение TCNT1<780. **Обратите внимание**, что в этом выражении мы тоже используем имя TCNT1. То есть проверяем значение всего шестнадцатиразрядного счетного регистра. Цикл проверки будет выполняться о тех пор, пока значение счетного регистра не будет превышать 780. Как только окажется, что это не так, цикл завершается, а с завершением цикла завершается и вся функция `wait1`.

Листинг 1.12

```
/*****************************************************
Project : Prog6
Пример 6
Бегущие огни (задержка с использованием таймера без использования прерываний)

Chip type           : ATtiny2313
Clock frequency     : 4,000000 MHz
Data Stack size     : 32
*****************************************************/

#include <tiny2313.h>

void wait1 (void)    // --------------- Функция задержки
{
    TCNT1=0;
    while (TCNT1<780) {};
}

void main(void)      // ----------------- Главная функция программы
{
unsigned char rab;   // Вводим переменную rab

PORTB=0xFF;    // Инициализация порта B
DDRB=0xFF;

PORTD=0x7F;    // Инициализация порта D
DDRD=0x00;

TCCR1A=0x00;   // Инициализация таймера/счетчика 1
TCCR1B=0x05;

ACSR=0x80;     // Инициализация аналогового компаратора

while (1)
    {
    if (PIND.0==1)  // Проверка состояния переключателя
          {                     // Сдвиг вправо
          rab = 0b10000000;     // Запись начального значения
          while (rab!=0)
               {
               PORTB=rab^0xFF;  // Запись в порт с инверсией
               rab = rab >> 1;  // Сдвиг разрядов
               wait1 ();        // Задержка в 200 мсек
               }
          }
    else
          {                     // Сдвиг влево
          rab = 0b00000001;     // Запись начального значения
          while (rab!=0)
               {
               PORTB=rab^0xFF;  // Запись в порт с инверсией
               rab = rab << 1;  // Сдвиг разрядов
               wait1 ();        // Задержка в 200 мсек
               }
          }
    };
}
```

1.8. Использование прерываний по таймеру

Постановка задачи

В **предыдущем примере** мы использовали таймер для формирования задержки, но не использовали его главного преимущества: способности вызывать прерывания. На практике подобным образом почти никогда не поступают. Чаще всего в подобных случаях применяют прерывания по таймеру. Это позволяет более точно формировать интервалы времени, но главное — позволяет разгрузить центральный процессор.

Пока таймер формирует задержку, программа может выполнять любые другие действия. В результате программу бегущих огней можно легко совместить, **например**, с программой генерации звуков. Но не будем усложнять нашу задачу и сформулируем ее следующим образом:

«Создать новую программу «бегущих огней» с использованием прерываний по таймеру».

Схема

Схему оставим без изменений (см. **рис. 1.11**).

Алгоритм

Поставленная выше задача потребует **полной переделки всей нашей программы**. Ведь изменится режим работы таймера. В данном конкретном случае удобнее всего использовать режим совпадения. Точнее, его подрежим «сброс при совпадении». В этом режиме таймер сам периодически вырабатывает запросы на прерывание с заранее заданным периодом.

Все функции управления «движением огней» выполняет процедура обработки прерывания. При каждом вызове прерывания процедура производит сдвиг «огней» на один шаг в нужном направлении.

Для того, чтобы обеспечить такую же скорость движения «огней», как в предыдущем примере, мы должны использовать те же самые коэффициенты деления. Для начала необходимо включить предварительный делитель и выбрать для него коэффициент деления 1/1024.

Второй коэффициент деления (780) мы помещаем в специальный системный регистр — **регистр совпадения**. Сравнение содержимого счетного регистра с содержимым регистра совпадения будет происходить на аппаратном уровне. В **режиме «сброс при совпадении»** таймер работает следующим образом. Сразу после запуска значение счетного регистра начнет увеличиваться. Когда это значение окажется равным значению регистра совпадения, таймер автоматически сбросится и продолжит работу с нуля. В момент сброса таймера формируется запрос на прерывание.

Для имитации бегущих огней, как и в предыдущих примерах, мы будем использовать **операции сдвига**. При этом нам также понадобится специальный рабочий регистр. То есть один из регистров общего назначения, в котором будет храниться текущее состояние наших «огней». В начале программы в рабочий регистр необходимо записать исходное значение. То есть число, один из разрядов которого равен единице, а остальные — нулю. В результате операций сдвига эта единица будет перемещаться вправо или влево, создавая эффект бегущего огня. Проверка состояния кнопки и сдвиг на один шаг будет производиться при каждом вызове процедуры обработки прерывания.

Исходя из вышесказанного, алгоритм работы программы состоит из двух независимых алгоритмов. **Во-первых**, это алгоритм основной программы, а **во-вторых**, алгоритм процедуры обработки прерывания. Рассмотрим их по порядку.

Алгоритм основной программы:

1. Настроить стек и порты ввода—вывода микроконтроллера;
2. Настроить таймер и систему прерываний;
3. Записать в рабочий регистр исходное значение;
4. Разрешить работу таймера;
5. Разрешить прерывания;
6. Перейти к выполнению основного цикла.

Так как все операции, связанные с движением огней, выполняет процедура обработки прерываний, в основном цикле программы нам ничего делать не нужно. Для простоты оставим основной цикл пустым.

Алгоритм процедуры обработки прерывания:

1. Проверить состояние переключателя режимов;

2. Если контакты переключателя разомкнуты, произвести сдвиг всех разрядов рабочего регистра на один разряд вправо. Если в результате этого сдвига единичный бит выйдет за пределы байта, создать новый единичный бит в крайней левой позиции;
3. Если контакты переключателя замкнуты, произвести сдвиг всех разрядов рабочего регистра на один разряд влево. Если в результате этого сдвига единичный бит выйдет за пределы байта, создать новый единичный бит в крайней правой позиции;
4. Вывести содержимое рабочего регистра в порт РВ, предварительно проинвертировав его;
5. Закончить процедуру обработки прерывания.

Программа на Ассемблере

Текст возможного варианта программы на языке Ассемблер приведен в **листинге 1.13**. В программе встречаются несколько новых для нас операторов.

Обратите внимание. *Используется новый для нас флаг — **флаг глобального разрешения прерываний**, который называется I.*

Мы уже упоминали этот флаг в **главе 3**. Флаг I, так же, как флаги С и Z, является одним из разрядов регистра SREG. Однако управление флагом I происходит совсем по-другому. На него не влияют ни арифметические, ни логические операции, а тем более операции сравнения. Для установки и сброса этого флага в системе команд предусмотрены **две специальные команды** (описаны ниже). Если флаг I сброшен, то все прерывания в микроконтроллере запрещены. Если флаг установлен, работа системы прерываний разрешается. Рассмотрим теперь по порядку все новые для нас операторы.

.dseg

Оператор выбора сегмента памяти данных. До сих пор во всех предыдущих ассемблерных программах мы обязательно использовали оператор `.cseg`, который позволял нам выбирать программный сегмент памяти. Пора научиться работать и с другими сегментами.

Следующий по значению после программного сегмента — это **сегмент памяти данных**, то есть сегмент ОЗУ. В программе (**листинг 1.13**) в **строке** 6 производится выбор именно этого сегмента.

.byte

Оператор резервирования памяти. Это один из операторов, которые действуют в сегменте памяти данных. Оператор позволяет зарезервировать один или несколько байтов (ячеек ОЗУ) для того, чтобы затем использовать их в программе. Вы спросите: зачем это нужно? Основная цель резервирования — учет и распределение памяти.

Если программист будет произвольно, по своему усмотрению, выбирать адреса ячеек ОЗУ для той либо иной задачи, то ему придется внимательно следить за тем, чтобы не выбрать повторно одну и ту же ячейку для хранения разных значений. Иначе программа при записи одного значения испортит второе, что приведет к ошибке в ее работе.

Механизм резервирования памяти позволяет транслятору контролировать использование памяти и исключать двойное использование ячеек. Кроме того, подобный механизм вообще избавляет программиста от необходимости запоминать адреса. Все происходит автоматически.

Оператор `.byte` имеет всего один параметр — **количество ячеек, которые нужно зарезервировать**. В нашей программе применяется лишь одна команда, резервирующая память (**строка 8, листинг 1.13**). В данном случае резервируется всего одна ячейка памяти. Метка `buf`, поставленная перед оператором, используется для обращения к зарезервированной ячейке.

reti

Оператор завершения подпрограммы обработки прерывания. Действие этого оператора аналогично действию оператора `ret`. Он извлекает адрес из стека и передает управление по этому адресу. Различие состоит в том, что команда `reti` еще и устанавливает в единицу флаг глобального разрешения прерываний I.

sts

Команда записи содержимого РОН в ОЗУ. Имеет два параметра. Первый параметр — **адрес ячейки памяти**, куда записываются данные.

Второй параметр — **имя регистра** источника данных. В **строке 49** программы содержимое регистра `rab` записывается в ОЗУ по адресу, определяемому меткой `buf`.

lds

Команда чтения информации из ячейки памяти. Прочитанная информация записывается в один из РОН. Команда также имеет два параметра. Первый параметр — **имя РОН,** куда записываются считанные данные. Второй параметр — **адрес ячейки памяти (источника данных).**

sei

Команда разрешения прерываний. Эта команда устанавливает флаг I. Тоесть разрешает все прерывания.

Листинг 1.13

```
    ;###############################################
    ;##               Пример 7                    ##
    ;##            "Бегущие огни"                 ##
    ;##  с использованием прерываний от таймера   ##
    ;###############################################

    ;------------------------ Псевдокоманды управления

1   .include "tn2313def.inc"         ; Присоединение файла описаний
2   .list                            ; Включение листинга

3   .def  temp = R16                 ; Определение главного рабочего регистра
4   .def  rab = R17                  ; Определение рабочего регистра для команд сдвига

5   .equ  kdel = 780

    ;------------------------ Резервирование ячеек памяти

6             .dseg                  ; Выбираем сегмент ОЗУ
7             .org    0x60           ; Устанавливаем текущий адрес сегмента

8   buf:      .byte   1              ; Один байт для хранения рабочего значения

    ;------------------------ Начало программного кода

9             .cseg                  ; Выбор сегмента программного кода
10            .org    0              ; Установка текущего адреса на ноль

    ;------------------------ Переопределение векторов прерываний

11  start:    rjmp    init           ; Переход на начало программы
12            reti                   ; Внешнее прерывание 0
13            reti                   ; Внешнее прерывание 1
14            reti                   ; Таймер/счетчик 1, захват
15            rjmp    prtim1         ; Таймер/счетчик 1, совпадение, канал A
16            reti                   ; Таймер/счетчик 1, прерывание по переполнению
17            reti                   ; Таймер/счетчик 0, прерывание по переполнению
18            reti                   ; Прерывание UART прием завершен
19            reti                   ; Прерывание UART регистр данных пуст
```

```
            reti                        ; Прерывание UART передача завершена
            reti                        ; Прерывание по компаратору
            reti                        ; Прерывание по изменению на любом контакте
            reti                        ; Таймер/счетчик 1. Совпадение, канал B
            reti                        ; Таймер/счетчик 0  Совпадение, канал B
            reti                        ; Таймер/счетчик 0  Совпадение, канал A
            reti                        ; USI готовность к старту
            reti                        ; USI Переполнение
            reti                        ; EEPROM Готовность
            reti                        ; Переполнение охранного таймера

; ------------------ Модуль инициализации
init:
;------------------------- Инициализация стека

            ldi     temp, RAMEND        ; Выбор адреса вершины стека
            out     SPL, temp           ; Запись его в регистр стека

;------------------------- Инициализация портов ВВ

            ldi     temp, 0             ; Записываем ноль в регистр temp
            out     DDRD, temp          ; Записываем этот ноль в DDRD (порт PD на ввод)

            ldi     temp, 0xFF          ; Записываем число $FF в регистр temp
            out     DDRB, temp          ; Записываем temp в DDRB (порт PB на вывод)
            out     PORTB, temp         ; Записываем temp в PORTB (потушить светодиод)
            out     PORTD, temp         ; Записываем temp в PORTD (включаем внутр.резист.)

;------------------------- Инициализация таймера T1

            ldi     temp, 0x0D          ; Выбор режима таймера
            out     TCCR1B, temp
            ldi     temp, high(kdel)    ; Старший полубайт кода совпадения
            out     OCR1AH, temp        ; Запись в регистр совпадения старш  полубайта
            ldi     temp, low(kdel)     ; Младший полубайт кода совпадения
            out     OCR1AL, temp        ; Запись в регистр совпадения младш  полубайта

;-------------------------- Определение маски прерываний

            ldi     temp, 0b01000000    ; Байт маски. Разрешено одно прерывание (№4)
            out     TIMSK, temp         ; Записываем маску

;-------------------------- Инициализация компаратора

            ldi     temp, 0x80          ; Выключение компаратора
            out     ACSR, temp

;------------------------- Начало основной программы

main:       ldi     rab, 0b00010000     ; Запись начального значения
            sts     buf, rab            ; Запись содержимого регистра rab в ОЗУ

            sei                         ; Разрешение прерываний
m1:         rjmp    m1                  ; Пустой бесконечный цикл

;=====================================================
;       Подпрограмма обработки прерываний
;=====================================================

prtim1:     push    temp                ; Сохраняем регистр temp
            push    rab                 ; Сохраняем регистр rab

            lds     rab, buf            ; Читаем содержимое rab из ОЗУ
            in      temp, PIND          ; Считываем содержимое порта PD
            sbrs    temp, 0             ; Проверка младшего разряда
            rjmp    p2                  ; Если не ноль, переходим к сдвигу влево

;------------------------ Сдвиг вправо

p1:         lsr     rab                 ; Сдвиг содержимого рабочего регистра
            brcc    p3                  ; Если не дошло до конца регистра пропустить
            ldi     rab, 0b10000000     ; Запись начального значения
            rjmp    p3                  ; В конец

;------------------------ Сдвиг влево
```

```
62  p2:     lsl    rab              ; Сдвиг содержимого рабочего регистра
63          brcc   p3               ; Если не дошло до конца регистра пропустить
64          ldi    rab, 0b00000001  ; Запись начального значения

    ;------------------------- Конец процедуры обработки прерывания

65  p3:     ldi    temp, 0xFF       ; Запись в temp числа $FF
66          eor    temp, rab        ; Инверсия содержимого rab (исключающее ИЛИ)
67          out    PORTB, temp      ; Вывод текущего значения в порт PB

68          sts    buf, rab         ; Запись регистра rab в ОЗУ

69          pop    rab              ; Восстанавливаем регистр rab
70          pop    temp             ; Восстанавливаем регистр temp

71          reti
```

Описание программы (листинг 1.13)

Начало программы (**строки 1—5**) у вас вызывать затруднений не должно. Здесь выполняется присоединение библиотечного файла, описание двух переменных (`temp` и `rab`) и описание константы `kdel`. Подобные операции мы уже выполняли в предыдущей программе. Различия начинаются в **строке 6**.

Тут мы впервые сталкиваемся с **процедурой резервирования ячеек ОЗУ**. Правда, зарезервируем мы для начала всего одну ячейку. Процесс резервирования похож на процесс автоматического размещения команд в программной памяти (см. **раздел 1.2**). Здесь также используется указатель текущего адреса. При резервировании ячеек указатель перемещается от нулевого адреса вверх, в сторону увеличения адресов. Если очередная директива `byte` резервирует N ячеек памяти, то и указатель перемещается на N позиций.

В нашей программе весь процесс резервирования занимает всего три **строки (строки 6—8)**:

- в **строке 6** выбирается нужный нам сегмент памяти (сегмент памяти данных);
- в **строке 7** выбирается новое значение для указателя в этом сегменте;
- в **строке 8** происходит собственно резервирование.

Так как в **строке 7** указателю присваивается значение 0x60, то именно по этому адресу будет располагаться ячейка памяти, резервируемая в **строке 8**. Почему же мы выбрали такой адрес?

Вспомните **схему распределения памяти микроконтроллера AVR [3]**. Ячейки ОЗУ с адресами от 0 до 0x1F совмещены с файлом регистров общего назначения, ячейки с адресами 0x20—0x5F совмещены с регистрами ввода—вывода. Ячейка с адресом 0x60 — это первая ячейка ОЗУ, предназначенная исключительно для хранения данных.

Зарезервированная нами ячейка далее в программе будет использоваться в качестве буфера для хранения содержимого рабочего регистра `rab` в промежутке между двумя вызовами прерывания. Именно из этих соображений для нее выбрано имя `buf`.

Резервированием памяти заканчивается модуль определений. Далее начинается **непосредственно программный код**, то есть код, помещаемый в программную память. Поэтому мы выбираем программный сегмент памяти (**строка 9**).

В **строке 10** устанавливается начальное значение указателя для этого сегмента. Далее начинается код самой нашей программы. Но начинается код программы совсем не так, как мы уже привыкли во всех предыдущих примерах. **Строки 11—29** занимает **блок команд переопределения векторов прерываний**. До сих пор в наших программах мы не имели подобного блока команд, потому что до сих пор мы не использовали прерываний.

Напомню **определение**: ***векторами прерываний*** *называется несколько специально зарезервированных адресов в начале программной памяти, предназначенных для обслуживания прерываний.*

Микроконтроллер ATiny2313 имеет таблицу векторов прерываний, состоящую из 19 адресов (с адреса 0x0000 по адрес 0x0012). Каждый из этих адресов, по сути, является адресом начала процедуры обработки одного из видов прерываний. Переопределение векторов состоит в том, что в каждую такую ячейку мы можем поместить команду безусловного перехода, передающую управление на адрес в программной памяти, где уже действительно начинается соответствующая процедура.

Обычно программа не использует сразу все заложенные в микропроцессор прерывания. **Например**, в нашем случае используется лишь одно прерывание — **прерывание по совпадению таймера**. Поэтому переопределение производят только для тех векторов, которые используются в данной программе. Однако и все остальные векторы

принято не оставлять без внимания. По всем остальным адресам таблицы принято ставить команды-заглушки.

Назначение команды-заглушки: предотвратить негативные последствия в случае ошибочного вызова незадействованного прерывания. Иногда в качестве такой заглушки применяют безусловный переход по нулевому адресу. Но удобнее всего использовать команду завершения процедуры обработки прерывания (`reti`). Если ненужное нам прерывание все же сработает, то оно тут же завершится, не нанеся никакого урона.

Какие же вектора прерываний переопределяются в нашей программе? **Во-первых, вектор нулевого адреса**. По адресу 0x0000 (**строка 11** программы) помещается команда безусловного перехода по метке `init`. В **строке** с этой меткой начинается основная процедура нашей программы. Как известно, ***нулевой адрес*** *— это вектор начального сброса микроконтроллера*. Именно с этого адреса начинается выполнение программы после системного сброса. Безусловный переход с нулевого адреса позволяет «перепрыгнуть» таблицу векторов прерываний и разместить основную программу за пределами этой таблицы.

Второй переопределяемый вектор — это вектор прерываний по совпадению таймера/счетчика T1. Его адрес равен 0x0004. Сюда мы помещаем команда безусловного перехода на метку `prtim1` (**строка 15** программы). Именно с этой метки начинается процедура обработки данного прерывания.

По всем остальным адресам таблицы помещены команды `reti`.

Сразу за таблицей векторов прерываний начинается **модуль инициализации**. Подобный модуль нам не в новинку. Модуль инициализации обязательно входит в любую программу. Наша программа — это всего лишь новый вариант программы для уже знакомой нам схемы бегущих огней. Режимы работы большинства систем микроконтроллера не изменяются. Поэтому модуль инициализации новой программы почти полностью повторяет соответствующий модуль из предыдущего примера. В предыдущей программе (**листинг 1.11**) подобный модуль занимал **строки 8—19**.

Но есть и **отличия**. В новой программе немного по-другому происходит инициализация таймера/счетчика. Теперь таймер должен быть переведен в режим сброса при совпадении. Возможны два варианта реализации такого режима:

- сброс при совпадении в канале А;
- сброс при совпадении в канале В.

Для каждого из каналов имеется свой собственный регистр совпадения. Не буду вдаваться в подробности. Просто скажу, что **мы выберем канал А**. Для того, чтобы перевести наш таймер/счетчик в выбранный нами режим, достаточно в регистр конфигурации таймера TCCR1B записать код 0x0D (**строки 38, 39**). Этот код не только переводит таймер в выбранный нами режим, но и устанавливает коэффициент предварительного деления, равный 1/1024. Подробнее о конфигурации таймера/счетчика смотрите в **главе 6**.

После того, как режим таймер выбран, нужно **записать код совпадения** в соответствующий регистр. Для канала А этот регистр называется OCR1A. Он имеет шестнадцать разрядов и физически состоит из двух отдельных регистров OCR1AH и OCR1AL. В каждую из этих половинок регистра записывается своя часть кода совпадения. В регистр OCR1AH записывается старший байт (**строки 40, 41**), а в регистр OCR1AL — младший байт (**строки 42, 43**) кода. Данный регистр совпадения обладает **свойством двойной буферизации**. Поэтому и здесь важен порядок записи двух его половинок. Сначала нужно записывать старший байт кода, а затем младший.

После инициализации таймера необходимо **инициализировать систему прерываний**. Инициализация системы прерываний сводится к выбору нового значения маски прерываний по таймеру. Значение маски записывается в регистр TIMSK. В данном случае нам нужно разрешить лишь один вид прерываний: прерывания по совпадению в канале А. Для этого соответствующий выбранному прерыванию бит в байте маски должен быть установлен в единицу.

Остальные биты должны оставаться равными нулю. Запись маски производится в **строках 44 и 45**. Во всем остальном новый модуль инициализации полностью соответствуют аналогичному модулю в программе из предыдущего примера (**листинг 1.11**).

За модулем инициализации начинается **основная программа**. В нашем случае она занимает всего четыре **строки** (**строки 48—51**). В **строках 48, 49** происходит присвоение начального значения рабочему регистру `rab` и сохранение этого значения в буфере `buf`. Как и в предыдущих примерах, рабочий регистр будет использоваться для операций сдвига, имитирующих движение нашего «огня».

Начальное значение должно представлять собой двоичное число, один двоичный разряд которого равен единице, а все остальные — нулю. Затем процедура обработки прерывания будет двигать этот разряд вправо и влево. Поэтому будет логично, если первоначально нашу единичку мы расположим где-то посередине. То есть выберем в

качестве начального значения, **например**, число 0b00010000. Что и сделано в **строке 48**.

Так как мы договорились, что в промежутках между двумя прерываниями эта величина будет храниться в буфере `buf`, в **строке 49** содержимое `rab` помещается в этот буфер. Теперь все готово к запуску системы прерываний. В **строке 50** находится команда, разрешающая все прерывания. **Обратите внимание**, что к этому моменту наш таймер/счетчик уже находится в режиме счета. Он начал работать сразу после записи значения в регистр TCCR1B. Однако все прерывания до сих пор были запрещены. Теперь, когда прерывания мы разрешили, система бегущих огней сразу начинает работать.

В **строке 51** основная программа завершается. Так как все операции по управлению движением «огней» выполняет процедура обработки прерывания, то основной программе больше ничего делать не нужно. Поэтому в **строке 51** организован бесконечный цикл. Он представляет собой безусловный переход сам на себя. Попав в такой цикл, программа будет бесконечно выполнять один и тот же оператор.

Немного о том, **как происходит вызов прерывания**. Таймер/счетчик непрерывно производит подсчет тактовых импульсов системного генератора. В момент, когда содержимое счетного регистра совпадет с содержимым регистра OCR1A, счетчик сбрасывается и начинает счет сначала. При очередном совпадении все повторяется. В момент сброса счетчика вызывается прерывание. Таким образом, процедура обработки прерывания выполняется периодически, каждый раз, когда счетчик досчитает до момента совпадения.

Коэффициент предварительного деления и величину кода совпадения мы выбрали таким образом, что период, с которым происходит вызов прерывания, равен 200 мс. То есть соответствует нашему техническому заданию. Процедура обработки прерывания заканчивается гораздо быстрее. Время выполнения этой процедуры примерно равно 6 мкс. Поэтому к тому времени, когда прерывание будет вызвано повторно, процедура обработки предыдущего прерывания уже давно закончится.

Теперь **перейдем к самой процедуре обработки прерывания**. Текст этой процедуры занимает **строки 52—71**. Начинается процедура с сохранения всех регистров, которые она в дальнейшем будет использовать (**строки 52, 53**). Как видите, мы сохраняем даже регистр `rab`. Теперь, в случае необходимости, наша основная программа сможет использовать этот регистр для своих целей. Так как в промежутке между двумя прерываниями содержимое `rab` хранится в буфере ОЗУ, то в **строке 54** мы извлекаем это значение из буфера и помещаем в `rab`.

Теперь все готово к **операции сдвига**. Но сначала нам нужно определить направление этого сдвига. Для этого достаточно проверить состояние контактов переключателя. Проверка производится в **строках 55—57**. В **строке 55** читается содержимое порта PD и записывается в регистр `temp`. В **строке 56** проверяется значение младшего разряда считанного значения. Если значение этого разряда равно единице (контакты переключателя разомкнуты), то **строка 57** пропускается, и программа переходит к процедуре сдвига вправо, которая начинается в **строке 58**. Если контакты замкнуты, то выполняется безусловный переход в **строке 57**, и управление передается по метке `p2`, где начинается процедура сдвига влево.

Процедура сдвига вправо занимает **строки 58—61**. Собственно сдвиг происходит в **строке 58**. В **строке 59** происходит проверка, не дошла ли в результате этого сдвига сдвигаемая единица до последнего разряда. Так же, как и в предыдущих примерах, признаком достижения конечной позиции служит появление единицы в флаге переноса (вспомните **табл. 1.2**).

Проверка флага переноса производится в **строке 59**. Если значение флага равно нулю, **строка 60** программы пропускается. Если же значение флага окажется равным единице, то команда в **строке 60** записывает в регистр `rab` новое значение. После записи этого значения единица окажется в самом старшем разряде. Таким образом организуется движение единицы по кругу (дойдя до крайней правой позиции, единица появляется слева).

В **строке 61** процедура сдвига вправо завершается. Управление передается по метке `p3`. То есть к процедуре вывода сдвинутого значения в порт.

Процедура сдвига влево (**строки 62—64**) работает аналогично предыдущей процедуре. Отличие состоит лишь в том, что здесь применяется другая команда сдвига (**строка 62**). Кроме того, при достижении крайней позиции регистру присваивается другое начальное значение. Теперь единица окажется в самом младшем разряде. Таким образом организуется кольцевое движение, но в другую сторону.

В **строке 65** начинается процедура вывода содержимого `rab` в порт PB. Процедура занимает **строки 65—67**. Точно такая же процедура применялась и в двух предыдущих версиях программы бегущих огней.

В **строках 68—70** происходит подготовка к завершению процедуры обработки прерывания. Сначала содержимое `rab` сохраняется в буфере ОЗУ (**строка 68**). Затем в **строках 69, 70** восстанавливаются значения регистров `temp` и `rab`. И, наконец, в **строке 71** процедура обработки прерывания завершается.

Программа на языке СИ

Как мы убедились на примере Ассемблера, для нашей новой задачи потребуются довольно **значительные изменения программы**. При разработке программы средствами системы CodeVisionAVR в подобной ситуации целесообразнее воспользоваться **мастером**, при помощи которого удобно создать новую заготовку программы.

Для создания новой заготовки программы удобно восстановить настройки из созданного ранее примера, подкорректировать их в соответствии с новыми требованиями и записать под новым именем. Для этого зайдем в программу CodeVisionAVR и запустим мастер. Для запуска мастера достаточно нажать на панели инструментов кнопку .

После запуска мастера все его управляющие элементы будут находиться в исходном состоянии. То есть иметь **значения по умолчанию**. Теперь нам нужно восстановить настройки из самого первого примера (см. **раздел 1.2**).

Для этого в меню «File» мастера выберем пункт «Open», как показано на **рис. 1.12**. Появится стандартное окно открытия файла. Найдите на вашем диске файл Prog1.cwp и откройте его. После того, как вы его откроете, все органы управления во всех вкладках мастера примут те значения, какие они имели при создании программы Prog1 (см. **рис. 1.3—1.6**).

Теперь нужно **сохранить этот пример с другим именем**. Для этого при помощи пункта меню «File / Save As» сделайте копию только что загруженного файла. Копию поместите в новую директорию с именем Prog7. В этой директории будет храниться наш новый проект. Теперь можно приступать к изменениям настроек под новые требования.

Исходя из технического задания, нам придется **изменить лишь настройки нашего таймера**. Для этого сначала откройте вкладку «Timers». На этой вкладке вы увидите еще три вкладки поменьше (см. **рис. 1.13**). Эти вкладки предназначены для настройки двух системных таймеров (Timer0 и Timer1), а также охранного таймера (Watchdog). Открываем вкладку «Timer1», как показано на **рис. 1.13**. При помощи расположенных там элементов управления выставляем следующие настройки:

- **поле «Clock Source» (Источник Тактового Сигнала)** оставляем в положении «System Clock» (Тактовый сигнал от системного генератора).
- в **поле «Clock Value» (Значение тактовой частоты)** выбираем значение 3,906 кГц. На самом деле мы выбираем коэффициент

деления предварительного делителя. Но для удобства программиста показаны получаемые при этом частоты сигнала. Значения этих частот вычисляются исходя из выбранной частоты тактового генератора (см. вкладку «Chip»). Как вы помните, тактовая частота в нашем случае равна 4 МГц.

- в **поле «Mode»** выбираем режим работы таймера. В нашем случае мы должны выбрать режим «CTC top=OCR1A» (режим сброса при совпадении с регистром OCR1A). Два поля «Out. A» и «Out. B» оставляем без изменений. При помощи этих полей можно включить и настроить параметры вывода сигнала совпадения на внешние онтакты микроконтроллера. В данном случае нам это не нужно.

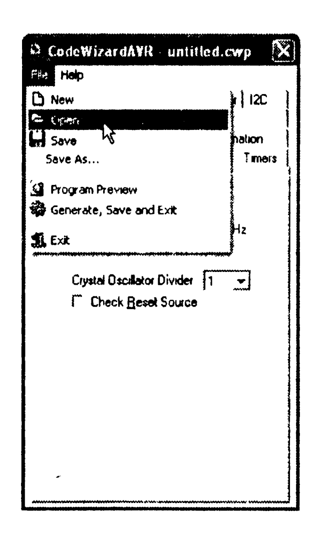


Рис. 1.12. *Загрузка параметров мастера.*

- **поле «Input Capt.» (Вход Захвата)** используется при работе в режиме захвата. Мы не используем режим захвата, поэтому и это поле тоже оставим без изменения.
- **поле «Interrupt On» (Прерывание От)** нам нужно изменить. Это поле предназначено для разрешения или запрета различных видов прерываний от таймера. В правой части поля имеются две маленькие кнопочки, при помощи которых вы можете листать его содержимое. В процессе перелистывания в окошке появляются названия видов прерываний.

Слева от названия каждого вида прерывания имеется **поле выбора («Check Box»)**, при помощи которого вы можете включить либо выключить данное прерывание. Путем перелистывания найдите прерывание, которое называется «Compare A Mach» (при совпадении с А). Поставьте «птичку» в соответствующем поле «Check Box». Теперь нужное прерывание будет включено.

- **поля «Value» и «Inp. Capture»** оставим без изменений. Эти поля предназначены для задания начальных значений счетного регистра и содержимого регистра захвата.
- и, наконец, **два поля под общим названием «Comp.»** позволяют выставить значения, записываемые в регистры совпадения А и В. Мы будем использовать регистр А. Запишем в поле «А:» значение нужного нам коэффициента деления (3D0). В данном случае допускается только шестнадцатиричный формат.

В результате всех этих манипуляций окно настройки таймера должно выглядеть так, как показано на **рис. 1.13**. Сохраните новое значение параметров при помощи меню «File / Save» мастера.

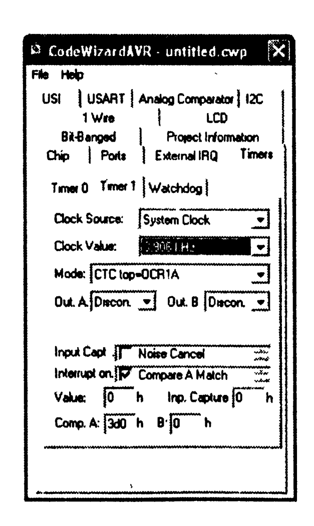


Рис. 1.13. *Настройка таймера*

Теперь можно **приступать к процессу формирования нового проекта**. Этот процесс был подробно описан в **разделе 1.1**, поэтому я повторяться не буду.

После того, как проект сформирован, приступаем **к созданию новой программы**. Возможный вариант такой программы приведен в **листинге 1.14**. В новой программе в основном использованы уже знакомые нам операторы. Из новых операторов появился лишь один.

#asm()

Это специальная системная функция, позволяющая в программе на СИ выполнять команды Ассемблера. Параметр функции — это строковая переменная или строковая константа, значение которой представляет собой текст команды на Ассемблере. Подобная функция добавляет программе значительную гибкость. **В строке 31** программы (**листинг 1.14**) при помощи функции `#asm()` выполняется ассемблерная команда разрешения прерывания `sei`. Команды такого уровня не имеют аналогов в традиционном синтаксисе языка СИ. Да и нецелесообразно выдумывать новые команды, когда удобнее просто выполнить команду Ассемблера.

Листинг 1.14

```
  /*****************************************************
  This program was produced by the
  CodeWizardAVR V1.24.4 Standard
  Automatic Program Generator
  © Copyright 1998-2004 Pavel Haiduc, HP InfoTech s.r.l.

  Project : Prog 7
  Comments: Бегущие огни с использованием прерываний

  Chip type           : ATtiny2313
  Clock frequency     : 4,000000 MHz
  Memory model        : Tiny
  External SRAM size  : 0
  Data Stack size     : 32
  *****************************************************/
1 #include <tiny2313.h>

  // Описание глобальных переменных
2 unsigned char rab;

  // Прерывание по совпадению таймера T1

3 interrupt [TIM1_COMP] void timer1_comp_isr(void)
      {
4     if (PIND.0==1)  // Проверка состояния переключателя
          {
5         rab = rab >> 1;  // Сдвиг разрядов
6         if (rab==0) rab = 0b10000000;  // Если дошло до конца
          }
7     else
          {
8         rab = rab << 1;  // Сдвиг разрядов
9         if (rab==0) rab = 0b00000001;  // Сдвиг влево
```

```
              }
10            PORTB=rab^0xFF;  // Запись в порт с инверсией
              }

11        void main(void)
          {
12            CLKPR=0x80; // Отключить деление частоты системного генератора
13            CLKPR=0x00;

14            PORTA=0x00;   // Инициализация порта A
15            DDRA=0x00;

16            PORTB=0xFF;   // Инициализация порта B
17            DDRB=0xFF;

18            PORTD=0x7F;   // Инициализация порта D
19            DDRD=0x00;

20            TCCR0A=0x00; // Инициализация таймера T0
21            TCCR0B=0x00;
22            TCNT0=0x00;
23            OCR0A=0x00;
24            OCR0B=0x00;

25            TCCR1A=0x00; // Инициализация таймера T1
26            TCCR1B=0x0D;
27            TCNT1H=0x00;
28            TCNT1L=0x00;
29            ICR1H=0x00;
30            ICR1L=0x00;
31            OCR1AH=0x03;   // Наш коэффициент деления (030CH = 780)
32            OCR1AL=0x0C;   //
33            OCR1BH=0x00;
34            OCR1BL=0x00;

35            GIMSK=0x00;    // Инициализация внешних прерываний
36            MCUCR=0x00;

37            TIMSK=0x40;    // Запись маски прерываний

38            USICR=0x00;    // Инициализация универсального последовательного интерфейса

39            ACSR=0x80;     // Инициализация аналогового компаратора

40            rab=0b00010000;  // Присвоение начального значение переменной rab
41            #asm("sei")      // Команде разрешения прерываний
42            while (1) {};
          }
```

Описание программы (листинг 1.14)

Текст программы, сформированный мастером, содержит две функции. Вернее, еще не функции, а их заготовки. **Во-первых,** это главная функция `main`, которая в уже готовой доработанной программе занимает **строки 11—42.**

Кроме главной функции, первоначальная автоматически сформированная программа содержит заготовку **еще одной функции — функции обработки прерывания. В строках 3—10** мы можем видеть ее в уже доработанном виде. В первоначальном виде функция `main` содержит

только **строки** инициализации (**строки 12—39**), а функция обработки (функция «timer1_comp_isr») вообще не содержит ни одного оператора.

Посмотрите внимательно на текст программы. Описание функции timer1_comp_isr (**строка 3** программы) отличается от всех встречавшихся нам до сих пор описаний. Слева от стандартного описания функции добавлены еще два дополнительных элемента. **Первый** — это слово Interrupt (прерывание). Это управляющее слово указывает транслятору на то, что данная функция является процедурой обработки прерывания. Вид прерывания, которое будет вызывать данную функцию, указывается сразу за словом Interrupt в квадратных скобках. Выражение Interrupt [TIM1_COMP] означает, что данная функция является процедурой обработки прерывания по совпадению таймера T1.

Имя функции обработки прерывания, автоматически данное мастером (timer1_comp_isr), не является обязательным. Если вы пожелаете разработать программу без участия мастера, вы можете выбрать имя для вашей функции по своему усмотрению. Вы можете также поменять имя и в нашей автоматически сформированной программе.

Функция обработки прерывания, как и функция main, не должна иметь параметров и не возвращает никаких значений. Поэтому перед именем функции и в круглых скобках всегда стоит слово void.

Теперь посмотрим, **что же мы изменили в программе вручную**:

- из автоматически сформированной программы были удалены все лишние комментарии, а вместо них были добавлены другие, на русском языке;
- в программу были внесены новые команды и операторы, реализующие нужные нам алгоритмы.

И первое, что нам пришлось сделать, — это создать переменную rab. Эта переменная нам будет нужна для операций сдвига, и использоваться она будет как в функции main, так и в функции обработки прерывания. То есть переменная должна быть глобальной. Поэтому описание этой переменной мы поместили вне обеих функций в самом начале программы (**строка 2**).

Функция main претерпела наименьшие изменения. Потребовалось лишь добавить команду присвоения начального значения переменной rab (**строка 40**) и команду разрешения прерываний (**строка 41**). Главный же цикл программы оставлен пустым, как и в программе на Ассемблере.

Доработанная подобным образом функция main работает точно так же, как основная программа в программе на Ассемблере. То есть после выполнения команд конфигурации и разрешения прерывания таймер/счетчик будет запущен в режиме сброса при совпадении. Каждые 200 мс он будет вызывать процедуру обработки прерывания, то есть функцию timer1_comp_isr.

Теперь посмотрим, **как была доработана функция timer1_comp_isr.** Основная часть доработок пришлась именно на нее. Новые команды, составляющие тело функции, занимают в программе **строки 4—10**. Функция обработки прерывания так же, как и соответствующая процедура в программе на Ассемблере, занимается сдвигом содержимого переменной rab на один шаг влево или вправо и выводом полученного значения в порт PB. Но перед тем, как выполнить сдвиг, нужно определить направление этого сдвига.

Для этого нужно проверить состояние переключателя. Для проверки состояния переключателя служит команда if (**строка 4**). Эта команда проверяет значение младшего разряда порта PD. Если значение разряда равно единице (контакты переключателя разомкнуты), то выполняется процедура сдвига на один бит вправо (**строки 5, 6**). Если младший бит PD равен нулю (контакты переключателя замкнуты), то выполняется процедура сдвига на один бит влево (**строки 8, 9**).

Рассмотрим подробнее процедуры сдвига. Сдвиг на один бит вправо выполняется в **строке 5**. Оператор if в **строке 6** проверяет, не дошла ли сдвигаемая единица до конца байта. Признаком того, что единица уже дошла до конца, является равенство переменной rab нулю. Если условие выполняется, то переменной rab будет присвоено значение 0b10000000. То есть дойдя до правого края, единица появляется слева. Таким образом реализуется эффект кругового движения единичного бита.

Обратите внимание, что в команде if в **строке 6** не используются фигурные скобки. Язык СИ допускает опускать фигурные скобки только в том случае, когда в них заключен всего один оператор.

Процедура сдвига на один бит влево (**строки 8, 9**) выполнен аналогичным образом. Я думаю, что тут вы легко разберетесь сами.

После выполнения одной из вышеописанных процедур сдвига производится запись содержимого переменной rab в порт PB с одновременным инвертированием этого содержимого (**строка 10**). Записанное в этой **строке** выражение нам уже хорошо знакомо. Оно применялось нами в обоих предыдущих примерах.

1.9. Формирование звука

Постановка задачи

В общем случае задача формирования звука не составляет большого труда. Достаточно взять за основу схему с мигающим светодиодом (см. **раздел 1.5**), подключить вместо светодиода звуковой излучатель (например, телефонный капсюль), а в соответствующей программе (**листинг 1.7**) поменять константу задержки таким образом, чтобы частота «мигания» повысилась и достигла звукового диапазона.

Диапазон частот, которые может услышать человек, лежит в пределах примерно от 50 Гц до 15 кГц. Светодиод в упомянутой выше программе мигает с частотой 4 Гц. Если уменьшить время задержки в 1000 раз, то можно получить частоту сигнала на выходе, равную 4 кГц. Эта частота как раз входит в звуковой диапазон.

Предлагаемый выше способ формирования звукового сигнала реализует эту задачу программным путем. Однако для формирования звука гораздо удобнее использовать таймеры/счетчики микроконтроллера. **Попробуем создать простейшее сигнальное устройство, которое при нажатии разных клавиш будет издавать звуки разной частоты.**

Допустим, мы имеем семь кнопок (датчиков). Сформулируем задачу следующим образом:

«Разработать электронное устройство, имеющее семь входов и один звуковой выход. К каждому из входов подключен датчик, состоящий их двух нормально разомкнутых контактов. При замыкании контактов любого из датчиков устройство должно вырабатывать звуковой сигнал определенной частоты. Каждому датчику должна соответствовать своя собственная частота звукового сигнала. Если контакты всех датчиков разомкнуты, звуковой сигнал, а выходе должен отсутствовать. Назовем наше устройство Сигнализатор «Семь нот».

Схема

Поставленная выше задача прекрасно решается при помощи уже известного нам микроконтроллера Atiny2313. Выберем его и на этот раз. Микроконтроллер имеет два встроенных таймера/счетчика. Какой же из таймеров использовать нам? Для формирования звука лучше подходит шестнадцатиразрядный таймер. Чем больше разрядов, тем с большей точностью можно выбирать его коэффициент деления.

Это очень **важно для создания нотного стана**. Поэтому для формирования звука выберем шестнадцатиразрядный таймер T1. Теперь

определимся **с режимом работы** нашего таймера. Как и в случае с бегущими огнями, для генерации звука удобнее всего использовать режим СТС (сброс по совпадению). Нам просто нужно выбрать такой коэффициент деления, чтобы на выходе таймера получить колебания в звуковом диапазоне частот.

Прежде всего, нам нужно отказаться от предварительного деления. Если частота кварцевого генератора и код, помещаемый в регистр совпадения, останутся такими же, как в предыдущем примере (в программе «Бегущие огни»), то в новом варианте частота повысится более чем в тысячу раз и как раз попадет в нужный нам диапазон.

Теперь определимся с тем, как наш **сигнал будет попадать на внешний вывод микроконтроллера**. Конечно, это можно сделать программно, при помощи процедуры обработки соответствующего прерывания. Но микроконтроллер предусматривает прямой вывод сигнала на один из своих выходов. Причем предусмотрены отдельные выходы для каждого из каналов совпадения. Для канала А подобный выход называется OC1A. Он совмещен с третьим разрядом порта PB и является альтернативной функцией данного контакта.

Подключение и отключение сигнала совпадения к внешнему выводу OC1A производится программным путем. Это позволяет программе в нужный момент включать или выключать звук. Так как для вывода звука мы будем использовать один из разрядов порта PB, то для подключения датчиков воспользуемся другим портом. А именно портом PD. Вариант принципиальной схемы описанного выше устройства показан на **рис. 1.14**.

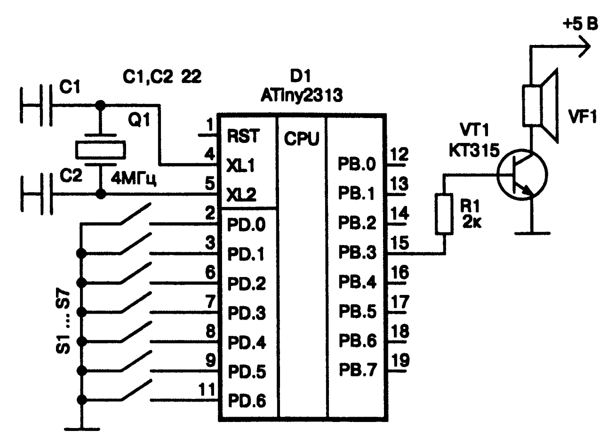


Рис. 1.14. *Схема сигнализатора «Семь нот»*

Как видно из рисунка, мы снова применили внешний кварцевый резонатор (Q1), естественно, не забыв при этом цепи согласования (C1, C2). При подключении датчиков используется та же схема, что использовалась до сих пор для подключения контактов переключателя. Датчики подключаются ко всем разрядам порта PD. При этом для правильной работы датчиков для каждого разряда порта PD должны быть активизированы встроенные резисторы нагрузки.

Для подключения звукоизлучателя (динамика) применяется ключевой каскад на транзисторе VT1. Это самый простой способ получить звук достаточной громкости, учитывая, что наш сигнал — это прямоугольные импульсы с амплитудой, почти равной напряжению питания. Транзисторный каскад нужен лишь для повышения нагрузочной способности.

Однако подобная схема имеет и свой **недостаток**. В отсутствии звукового сигнала на выходе 15 микроконтроллера обязательно нужно установить низкий логический уровень. Высокий логический уровень приведет к тому, что транзистор VT1 будет постоянно открыт. Это вызовет недопустимо большой ток через головку VF1. Постоянно протекающий ток через обмотку динамика вызовет излишнюю потерю мощности и может даже вызвать выход из строя как транзистора, так и динамика. При составлении программы мы должны учесть этот момент.

Алгоритм

На первый взгляд алгоритм такого устройства очень простой. При замыкании контактов любого из датчиков микроконтроллер должен загрузить в регистр совпадения нужный коэффициент и подключить выход таймера к выводу OC1B. При размыкании контактов датчика микроконтроллер должен отключить сигнал от внешнего вывода OC1B и подать на него низкий логический уровень. Если контакты всех датчиков разомкнуты, то внешний вывод должен оставаться отключенным.

Однако схема построена таким образом, что ничто не мешает одновременно замкнуться сразу нескольким контактам. Возникает **вопрос**: что делать в этом случае? Самый правильный **ответ** — обеспечить систему приоритетов. При замыкании нескольких контактов программа должна реагировать лишь на один из них. На тот, приоритет которого выше.

Обычно в таких случаях используется следующий прием. Программа поочередно проверяет состояние всех датчиков, **например**, справа налево. Обнаружив первый же замкнутый контакт, программа прекращает сканирование и выдает звуковой сигнал, соответствующий этому датчику.

Договоримся, что датчику, подключенному к входу PD.0, будет соответствовать нота «До». Следующему датчику — нота «Ре», и так далее до ноты «Си». Коэффициенты деления для каждой из нот выбираются по законам музыкального ряда. Подробнее о выборе коэффициентов деления для синтезатора музыкального ряда можно узнать из [5].

Программа на Ассемблере

Возможный вариант программы на Ассемблере показан на **листинге 1.15**. В программе использованы следующие **новые для нас команды.**

brcs

Условный переход по признаку переноса. Выполняет передачу управления в случае, если признак переноса С равен единице. Данная команда является полной противоположностью уже знакомой нам команды `brcc`, которая, напротив, вызывает переход при отсутствии переноса.

add

Арифметическая операция сложения. Производит сложение содержимого двух РОН. Эта команда имеет два операнда, в качестве которых выступают имена складываемых регистров. В **строке 54** программы (**листинг 1.15**) к содержимому регистра `count` прибавляется содержимое регистра ZL. Результат помещается в ZL.

adc

Сложение с переносом. Этот оператор тоже выполняет сложение. Но в процессе сложения он учитывает перенос, возникший в предыдущей операции сложения. Команда сложения с учетом переноса используется при составлении программ, позволяющих складывать большие

числа. Если каждое из слагаемых занимает больше, чем один байт, то входящие в них байты складывают в несколько этапов.

Сначала складывают младшие байты, а затем старшие. При сложении младших байтов может возникнуть бит переноса (если результат оказался больше, чем 0xFF). Этот перенос и нужно учесть при сложении старших байтов. Команда adc производит сложение содержимого двух регистров, имена которых указаны в качестве ее операндов.

К полученной сумме добавляется значение признака переноса. В **строке 56** программы к содержимому регистра temp прибавляется содержимое регистра ZH с учетом значения признака переноса. Результат помещается в ZH.

clr

Сброс всех разрядов РОН. Команда имеет один параметр — имя РОН, разряды которого нужно сбросить. Действие этой команды равносильно записи в РОН числа 0x00.

mov

Передача данных между двумя РОН. Эта команда имеет два операнда. Первый операнд — имя регистра, получателя данных. Второй операнд — имя регистра-источника. Команда копирует содержимое одного регистра в другой.

lpm

Чтение байта данных из программной памяти. Микроконтроллеры AVR имеют отдельную память данных и отдельную память программ. Однако некоторые виды данных удобно хранить в памяти программ. К таким данным относятся наборы различных констант. В нашем случае в памяти программ удобно хранить набор коэффициентов деления для всех наших нот. Для извлечения данных из памяти программ используется команда lpm.

Хранение данных в программной памяти имеет свои **особенности**. Дело в том, что память программ состоит из набора шестнадцатиразрядных ячеек. Коды команд также имеют шестнадцать разрядов. Данные же нужно хранить в виде отдельных байтов. То есть в виде восьмиразрядных двоичных чисел.

Для того, чтобы эффективнее использовать программную память, она организована таким образом, что в каждой шестнадцатиразрядной ячейке программной памяти можно хранить два разных байта данных. Команда `lpm` может читать каждый такой байт по отдельности. Для этого используется альтернативная адресация. Благодаря альтернативной адресации, программная память в режиме чтении данных имеет в два раза больше ячеек, чем при чтении кодов команд.

Это нужно учитывать при использовании команды `lpm`. Если вы знаете адрес размещения данных согласно основного способа адресации, прежде, чем использовать его в команде `lpm`, ее значение необходимо умножить на два, чтобы получить адрес того же байта в альтернативной адресации. В команде `lpm` нет операнда, определяющего адрес ячейки, содержимое которой требуется прочитать. Этот адрес предварительно должен быть записан в регистровую пару Z.

Команда `lpm` имеет три модификации. Ниже приведен формат всех трех модификаций этой команды:

```
lpm
lpm   Rd, Z
lpm   Rd, Z+
```

Первая версия команды не имеет никаких операндов. Выполняя эту команду, микроконтроллер читает содержимое ячейки программной памяти, адрес которой записан в регистровой паре Z и помещает прочитанную информацию в регистр R0. Напоминаю, что в Z нужно помещать адрес ячейки в альтернативной адресации.

Вторая версия команды имеет два операнда:

- первый операнд — это имя регистра, куда будет помещен считанный байт;
- второй операнд всегда равен Z.

Новая версия команды работает так же, как предыдущая. Различие только в том, что прочитанная этой командой информация помещается в РОН, имя которого указано в качестве первого параметра.

Третья версия команды является модификацией второй. Она тоже имеет два операнда:

- первый операнд — это имя регистра, куда помещается прочитанный байт;
- второй операнд всегда равен Z+.

От второго варианта третий отличается тем, что сразу после чтения байта происходит автоматическое увеличение содержимого регистровой

пары Z на единицу. Данную команду удобно использовать для последовательного чтения ряда констант из программной памяти. При каждом последующем вызове команда будет читать следующую константу.

Первая модификация команды **использовалась** в старых версиях микроконтроллеров и оставлена для совместимости. В новых микроконтроллерах гораздо удобнее пользоваться второй и третьей модификациями.

.dw

Директива описания данных. При помощи этой директивы описываются данные, помещаемые в память программ или в энергонезависимую память. Оператор `.dw` описывает «слова» данных. То есть шестнадцатиричные числа, каждое из которых записывается в память в виде пары байтов. **В правой части**, сразу после оператора, помещается список чисел через запятую. При трансляции программы эти числа помещаются в программную память (или в EEPROM) одно за другим так же, как туда помещаются команды.

В строке 63 программы (**листинг 1.15**) в программную память помещается **набор коэффициентов деления**. Создается своеобразная **таблица** коэффициентов деления. Адрес начала таблицы соответствует метке `tabkd`. С точки зрения основной адресации, каждый коэффициент занимает одну шестнадцатиразрядную ячейку памяти.

С точки зрения альтернативной адресации, ***каждый коэффициент** — это два байта данных, записываемых в две соседние ячейки.* Причем сначала записывается младший байт, а затем старший. При чтении этих данных используется альтернативная адресация. Если мы будем последовательно читать таблицу, начиная с адреса `tabkd*2`, мы получим следующую цепочку данных:

0x8C, 0x12, 0x84, 0x10, 0xB8, 0x0E, 0xE4, 0x0D, 0x60, 0x0C, 0x36, 0x0B, 0xD2, 0x09.

Если вам не понятно, почему мы получим именно такую цепочку, вспомните, что первое число таблицы 4748 в шестнадцатиричном виде выглядит как 0x128C. То есть, его старший байт равен 0x12, а младший 0x8C. Шестнадцатиричное значение второго члена таблицы 4228 равно 0x1084. И так далее.

inc

Инкремент. Это очень простая команда. Она увеличивает содержимое одного из регистров общего назначения на единицу. У этой команды всего один параметр — имя регистра, содержимое которого нужно увеличить.

Листинг 1.15

```
;###############################################
;##               Пример 8                    ##
;##   Сигнальное устройство "Семь нот"        ##
;###############################################

;------------------------ Псевдокоманды управления

.include "tn2313def.inc"           ; Присоединение файла описаний
.list                              ; Включение листинга

.def     temp = R16                ; Определение регистра передачи данных
.def     count = R17               ; Определение регистра счетчика опроса клавиш

;------------------------ Начало программного кода

         .cseg                     ; Выбор сегмента программного кода
         .org     0                ; Установка текущего адреса на ноль

start:   rjmp     init             ; Переход на начало программы
         reti                      ; Внешнее прерывание 0
         reti                      ; Внешнее прерывание 1
         reti                      ; Таймер/счетчик 1, захват
         reti                      ; Таймер/счетчик 1, совпадение, канал А
         reti                      ; Таймер/счетчик 1, прерывание по переполнению
         reti                      ; Таймер/счетчик 0, прерывание по переполнению
         reti                      ; Прерывание UART прием завершен
         reti                      ; Прерывание UART регистр данных пуст
         reti                      ; Прерывание UART передача завершена
         reti                      ; Прерывание по компаратору
         reti                      ; Прерывание по изменению на любом контакте
         reti                      ; Таймер/счетчик 1. Совпадение, канал В
         reti                      ; Таймер/счетчик 0. Совпадение, канал В
         reti                      ; Таймер/счетчик 0. Совпадение, канал А
         reti                      ; USI готовность к старту
         reti                      ; USI Переполнение
         reti                      ; EEPROM Готовность
         reti                      ; Переполнение охранного таймера

; ------------------ Модуль инициализации

init:    ldi      temp, RAMEND     ; Инициализация стека
         out      SPL, temp

;------------------------- Инициализация портов В/В

         ldi      temp, 0x08       ; Инициализация порта PB
         out      DDRB, temp

         ldi      temp, 0x00
         out      DDRD, temp       ; Инициализация порта PD
         out      PORTB, temp      ; Выходное значение порта PB

         ldi      temp, 0x7F
         out      PORTD, temp      ; Включение внутренних резисторов

;------------------------- Инициализация компаратора
         ldi      temp, 0x80
```

```
36            out      ACSR, temp

       ;------------------------- Инициализация таймера/счетчика
37            ldi      temp, 0x09
38            out      TCCR1B, temp        ; Выбор режима
39     m1:    ldi      temp, 0x00
40            out      TCCR1A, temp        ; Отключение звука

       ;------------------------- Начало основного цикла программы

41     main:  clr      count               ; Обнуление счетчика опроса клавиш
42            in       temp, PIND          ; Чтение порта D
43     m2:    lsr      temp                ; Сдвигаем входной байт
44            brcc     m3                  ; Если текущий разряд был равен 0
45            inc      count               ; Увеличиваем показание счетчика
46            cpi      count, 7            ; Сравнение (7 - конец сканирования)
47            brne     m2                  ; Если не конец, продолжить
48            rjmp     m1                  ; Если не одна клавиша не нажата

49     m3:    lsl      count               ; Умножение номера кнопки на 2
50            mov      YL, count           ; Создаем первое слагаемое
51            ldi      YH, 0               ; Старший его байт равен нулю
52            ldi      ZL, low(tabkd*2)    ; Второе слагаемое - начало таблицы tabkd
53            ldi      ZH, high(tabkd*2)
54            add      ZL, YL              ; Складываем два 16-разр. слагаемых
55            adc      ZH, YH

56            lpm      YL, Z+              ; Читаем младший байт коэфф. деления
57            lpm      YH, Z               ; Читаем младший байт коэфф. деления
58            out      OCR1AH, r0          ; Вспомогательное значение
69            out      OCR1AL, r0          ; Записать в старш. часть регистра совпадения

60            ldi      temp, 0x40          ; Включить звук
61            out      TCCR1A, temp
62            rjmp     main

       ; ***********************************************************
       ; **          Таблица коэффициентов деления               **
       ; ***********************************************************

63     tabkd:  .dw     4748,4228,3768,3556,3168,2822,2514
```

Описание программы (листинг 1.15)

Четыре первые строки программы (**строки 1—4**), я думаю, пояснений не требуют. Все эти команды знакомы нам по предыдущему примеру. **Строки 5—25** занимает таблица переопределения векторов прерываний. Мы применили эту таблицу, несмотря на то, что данная программа не использует прерываний. Именно поэтому во всех ячейках таблицы, кроме ячейки с нулевым адресом, поставлены **команды-заглушки**. Для серьезных проектов применение таблицы считается обязательным. Это повышает надежность работы программы, а также повышает ее наглядность.

Строки 26—40 занимает **модуль инициализации**. Начинается он с инициализации стека (**строки 26, 27**). Затем производится инициа-

лизация портов ввода—вывода. Команды инициализации портов занимают **строки 28—34**.

Порт PB настраивается таким образом, что линия PB.3 работает в режиме вывода информации, а остальные линии — в режиме ввода. Все разряды порта PD настраиваются на ввод. В регистр PORTB записывается нулевое значение. При этом на выходе PB.3 появляется низкий логический уровень, закрывающий ключ VT1. В регистр PORTD записывается код 0x7F, который включает внутренние резисторы нагрузки.

Код 0x7F в двоичном виде выглядит как 0x01111111, поэтому он включает нагрузки для семи младших разрядов порта. Для старшего, восьмого разряда нагрузку включить невозможно, так как этот разряд просто отсутствует.

В **строках 35, 36** производится **инициализация компаратора**. И, наконец, в **строках 37—40** производится инициализация таймера T1. Сначала настраиваются его **режимы работы**. Для этого в регистр TCCR1B записывается код 0x09 (**строки 37, 38**). Этот код переводит таймер в режим CTC и выбирает коэффициент предварительного деления равным единице.

Затем в регистр TCCR1A записывается ноль (**строки 39, 40**). Как видите, в комментариях к этому действию написано: «выключение звука». В данном случае подобное утверждение справедливо. Одна из функций регистра TCCR1A — управление подключением сигнала от таймера на внешний выход OC1A, который нашем случае служит выходом звука.

Включением и отключением выхода OC1A управляет разряд номер 6 регистра TCCR1A. Единица в шестом разряде подключает таймер к выходу OC1A (включает звук). При нулевом значении шестого разряда выход OC1A отключается, а соответствующему контакту микросхемы возвращается его основная функция.

Он становится просто выводом порта PB, в который, как вы помните, записан логический ноль. Этот ноль появляется на выходе, закрывая ключ. Таким образом, при выключенном звуке на выходе всегда будет ноль. Из всего вышеизложенного вы уже поняли, что запись в регистр TCCR1A кода 0x00 равносильна отключению звука.

В **строке 41** начинается **основной цикл нашей программы**. В первой части цикла (**строки 41—48**) расположена процедура опроса датчиков. Для работы этой процедуры используется вспомогательный регистр `count`. Программа сканирует датчики один за другим, а регистр `count` используется для подсчета уже отсканированных датчиков.

Сканирование заканчивается тогда, когда обнаружится первый же датчик с замкнутыми контактами. При этом в регистре `count` останется номер этого датчика.

Рассмотрим работу процедуры сканирования подробнее. Перед началом сканирования содержимое регистра `count` обнуляется (**строка 41**). Затем производится чтение сигнала с контактов порта PD (**строка 42**). Считанный код помещается в регистр `temp`. Теперь код, находящийся в регистре `temp`, содержит полную информацию о состоянии всех семи датчиков.

Каждому из семи датчиков будет соответствовать один из семи младших разрядов кода в регистре `temp`:

- если в момент считывания контакты датчика были разомкнуты, то соответствующий разряд будет равен единице;
- если контакты датчика были замкнуты, соответствующий разряд будет равен нулю.

Сканирование датчиков сводится к проверке семи младших разрядов кода в регистре `temp`. Цикл сканирования составляют **строки 43—48**. В процессе сканирования программа просто сдвигает содержимое регистра `temp` вправо. В результате каждого сдвига содержимое очередного разряда попадает в флаг признака переноса.

По значению этого флага и определяется состояние датчика. Как только очередной разряд окажется равным нулю, это значит, что контакты соответствующего датчика были замкнуты. Поэтому цикл сканирования прекращается, и программа **приступает к процедуре формирования звука**.

Цикл сканирования повторяется семь раз (по количеству датчиков). Если после семи сдвигов нулевой бит не обнаружен, значит, контакты всех семи датчиков были незамкнуты. В этом случае управление передается по метке `m1`, где происходит выключение звука. Затем снова считывается состояние порта, и весь цикл сканирования повторяется сначала.

Логический сдвиг вправо разрядов регистра `temp` выполняется в **строке 43**. В **строке 44** производится проверка признака переноса. Если он равен нулю (контакты датчика замкнуты), то происходит переход к **строке 49** по метке `m3`.

Там **начинается вычисление параметров для формирования звука**. Если признак переноса равен единице (контакты датчика не замкнуты), цикл сканирования продолжается. Следующая команда (**строка 45**) увеличивает содержимое регистра `count`, осуществляя подсчет

датчиков. В **строке 46** содержимое count сравнивается с числом 7. Таким образом ограничивается количество проходов цикла сканирования.

Если содержимое count еще не достигло семи, то выполняется переход по метке m2, и цикл сканирования продолжается. Если же содержимое count окажется равным семи, то это означает, что все семь датчиков мы уже перебрали. В этом случае выполняется оператор безусловного перехода в **строке 48**, который передает управление по метке m1.

Теперь посмотрим, что же происходит, когда **процедура сканирования обнаружит сработавший датчик**. В этом случае управление передается к **строке 49** (метка m3). Регистр count к этому моменту содержит номер сработавшего датчика. Для датчика, подключенного к входу PD.0, этот код будет равен нулю. Для PD.1 код будет равен единице. И так далее.

Теперь нам нужно **сгенерировать звук**, соответствующий коду датчика. Для этого мы должны извлечь соответствующий коэффициент деления из программной памяти, поместить его в регистр совпадения и подключить сигнал с таймера на внешний выход.

Сначала займемся **извлечением коэффициента деления**. Как уже говорилось выше, все коэффициенты записаны в программную память и составляют таблицу коэффициентов деления (см. **строку 63**). Для извлечения нужного нам коэффициента воспользуемся командой lpm. Но сначала нам необходимо вычислить адрес соответствующей ячейки памяти.

Для этого вспомним, что *каждый коэффициент представляет собой два байта, записанные в две соседние ячейки памяти (по альтернативной адресации)*. Если адрес начала таблицы равен tabkd, то в альтернативной адресации он будет равен tabkd×2. Очевидно, что коэффициент деления для датчика номер ноль будет занимать ячейки с адресами tabkd×2 и tabkd×2+1. Коэффициент деления для датчика номер один мы найдем в ячейках tabkd×2+2 и tabkd×2+3. И так далее. В общем случае адрес ячейки, содержащей первый байт нужного нам коэффициента мы найдем по формуле

tabkd×2+Nd ×2,

где Nd — это номер датчика.

Отсюда наша **задача**: используя номер, записанный в регистре count, вычислить адрес ячейки, где хранится нужный коэффициент деления. Так как любой ***адрес** — это шестнадцатиразрядное двоичное*

число, нам придется производить операцию шестнадцатиразрядного сложения. В системе команд микроконтроллеров AVR такая команда отсутствует. Поэтому мы будем складывать шестнадцатиразрядные числа побайтно.

Команды вычисления адреса занимают **строки 49—55**. **В строке 49** происходит удвоение содержимого регистра `count` (умножение кода датчика на два). Для удвоения используется команда логического сдвига `lsl`.

Дело в том, что в двоичной системе логический сдвиг на один бит влево эквивалентен умножению на два. Теперь полученное в результате удвоения число необходимо прибавить к адресу начала таблицы. Для этого сформируем два шестнадцатиразрядных слагаемых. Первое слагаемое мы запишем в регистровую пару Y, а второе слагаемое — в регистровую пару Z. **Младший байт первого слагаемого** (удвоенный код датчика) мы берем из регистра `count` и помещаем в YL (**строка** 50). Так как датчиков всего семь (максимальное значение удвоенного кода равно 14), то **старший байт первого слагаемого** всегда будет равен нулю.

Запишем этот ноль в регистр YH (**строка 51**). В качестве второго слагаемого мы будем использовать число, равное удвоенному значению метки `tabkd`. Запишем младший и старший байты этого числа в регистровую пару Z (**строки 52, 53**). После того, как оба слагаемых сформированы, приступаем к процессу сложения. Сначала складываем младшие байты (**строка 54**). Затем складываем старшие байты с учетом переноса (**строка 55**). В результате сложения в регистре Z мы получим искомый адрес.

Теперь, используя этот адрес, приступим **к извлечению требуемого коэффициента деления**. В **строке 56** извлекается первый байт коэффициента. Причем используется версия команды `lpm`, которая автоматически увеличивает содержимое регистровой пары Z. Извлеченный байт помещается в младшую часть регистровой пары Y (регистр YL). Так как в содержимое Z стало на единицу большим, то очередная команда в **строке 57** извлекает очередной байт коэффициента деления. Извлеченный байт помещается в старшую часть регистровой пары Y (регистр YH).

В **строках 58 и 59** прочитанный только что коэффициент деления записывается в регистр совпадения OCR1A. При этом соблюдается **правило записи для регистров с двойной буферизацией:**

- сначала записывается старшая часть регистра (**строка 58**);
- затем записывается младшая (**строка 59**).

Сразу после записи коэффициента деления таймер начнет вырабатывать сигнал с нужной нам частотой. Теперь остается лишь подключить этот сигнал на выход (выполнить включение звука). Включение звука происходит в **строках 60, 61**.

Для этого в регистр TCCR1A записывается код 0x40. Этот код имеет единицу в шестом разряде, которая и подключает сигнал от таймера к выводу OC1A. В результате на выходе появляется звуковой сигнал, который через транзисторный ключ VT1 поступает на звуковой излучатель VF1.

Команда безусловного перехода в **строке 62** завершает основной цикл программы. Она передает управление по метке `main`. В результате весь описанный выше процесс повторяется. Если в результате нового опроса датчиков обнаружится, что их состояние не изменилось, программа просто подтвердит все настройки таймера и генерация звука не прервется. Если состояние датчиков изменилось, то изменится и частота звука. Если же при очередном сканировании контакты всех датчиков окажутся незамкнутыми, управление перейдет по метке `m1`, и звук прекратится.

Программа на языке СИ

Возможный вариант той же программы на языке СИ приведен в **листинге 1.16**. В программе применяются следующие новые для нас операторы.

for

Оператор цикла. Мы уже привыкли к оператору цикла `while`. Оператор `for` — это альтернативный способ организации циклов. В общем случае оператор `for` записывается следующим образом:

```
for (Выр1; Выр2; Выр3)
      {
            Тело цикла;
      }
```

Выр1, Выр2 и Выр3 — любые корректные выражения языка СИ. Каждое из этих выражений имеет свое определенное **назначение**.

Выр1 — это команда начальной установки. Она выполняется один раз перед началом цикла.

Выр2 — условие выполнения цикла. Обычно это логическое выражение. Значение Выр2 проверяется в начале каждого прохода. Пока

результат этого выражения «Истина» (не равен нулю), цикл продолжается. Как только результат Выр2 примет значение «Ложь» (станет равен нулю), цикл прекращается.

Выр3 — операция, выполняемая с параметром цикла. Это выражение выполняется в конце каждого прохода. Обычно в качестве Выр3 ставится команда, увеличивающая параметр цикла на единицу. Вот пример применения оператора `for`:

```
for (i=0; i<10; i++)
      {
            Тело цикла;
      }
```

Это простейший цикл с параметром `i`. Перед началом цикла параметру присваивается нулевое значение. Цикл выполняется до тех пор, пока `i` меньше десяти. Каждый раз после выполнения команд, составляющих тело цикла, оператор `i++` увеличивает значение переменной `i` на единицу. Выражение `i++` представляет собой одно из сокращений языка СИ. В развернутом виде та же команда выглядит так: `i=i+1`.

goto

Команда безусловного перехода. Тот, кто знаком с языком программирования Basic, хорошо знает эту команду. Команда `goto` в языке СИ то же самое, что `rjmp` на Ассемблере. Она имеет всего один параметр — имя метки. В **строке 19** нашей программы (**листинг 1.16**) команда `goto` передает управление к **строке 14** (по метке `m1`).

Кроме двух новых операторов, в нашей программе появляется новое для нас понятие: массив.

Определение. ***Массив** — это набор элементов, каждый из которых может иметь свое собственное значение. Все элементы массива всегда имеют один тип.*

Перед тем, как использовать массив, его, как и переменную, нужно описать. Описание массива очень похоже на описание переменной. В общем виде это выглядит следующим образом:

Тип Имя [Разм];

Как и в случае с переменной, сначала указывается тип массива, затем его имя:

- тип массива может принимать те же значения, что и тип переменной (см. **табл. 1.1**);
- имя массива выбирается в соответствии со стандартными правилами выбора имен.

В квадратных скобках указывается размер массива, то есть количество его элементов. В языке СИ при описании массивов необходимо обязательно указывать их размер, чтобы транслятор мог зарезервировать память для их размещения. Однако в любом месте программы размер массива допускается изменять. Это делается при помощи повторного описания массива, но уже с другим значением размера. Если новый размер окажется больше старого, массив просто пополнится новыми членами. При этом значения старых членов сохраняются. Если новый размер окажется меньше старого, все лишние члены удаляются. Значения удаленных членов массива будут утеряны.

Приведем **пример** описания массива:

```
int rabtab[5];
```

Описанный выше массив имеет имя `rabtab`, тип `int` и количество членов, равное пяти. Теперь посмотрим, как **применяются массивы**. С любым элементом описанного выше массива можно работать как с отдельной переменной. **Например**, можно присвоить ему значение:

```
rabtab[1] = 231;    // Первому члену массива присваивается
                    // значение 231.
```

Можно наоборот, значение одного из элементов массива присвоить переменной:

```
x = rabtab[8];      // Переменной x присваивается
                    //значение восьмого элемента массива.
```

Определение. *При использовании массива в тексте программы число в квадратных скобках уже означает **не размер** массива, а **номер элемента**, с которым производится данная операция. Поэтому такое число теперь называется **указателем массива**.*

Если размер массива равен N, то указатель массива может принимать значения от 0 до N-1. Если окажется, что значение указателя

выходит за указанные пределы, то транслятор выдаст сообщение об ошибке. В качестве указателя массива можно использовать не только числовую константу. В квадратные скобки вы можете вписать любую переменную или любое выражение. Главное, чтобы значение этого выражения входило в область допустимых значений.

Определение. *При описании массива можно одновременно производить его инициализацию. Под* ***инициализацией*** *понимается присвоение начальных значений всем элементам массива.*

Строка описания массива, производящая его инициализацию, выглядит примерно следующим образом:

```
int rabtab[5] = {23, 41, 52, 287, 40, 51};
char txt1[] = "Simferopol";
```

Начальные значения всех элементов записываются после знака «=» в фигурных скобках через запятую (числовые значения) либо в кавычках в виде символьной строки. В последнем случае в каждый элемент массива записывается код одного из символов.

Если при описании массива используется инициализация, допускается не указывать его размер. В этом случае размер определяется автоматически по количеству значений в фигурных скобках или по количеству символов.

Еще один аспект программирования на языке СИ, который мы будем впервые использовать в нашей программе, — это **директивы локализации переменных и массивов**. Так как наша версия СИ специально предназначена для составления программ для микроконтроллеров, она просто обязана иметь возможность явно указывать вид памяти, где нужно хранить ту либо иную переменную или массив.

По умолчанию транслятор сам решает эту задачу и размещает переменные и массивы по своему усмотрению. Но иногда вопрос размещения имеет принципиальное значение. В этом случае применяются **специальные директивы**. Эти директивы используются в процессе описания и указывают транслятору, в каком из видов памяти разместить данную конкретную переменную либо массив. **Например**, если вы желаете, чтобы значение вашей переменной хранилось в одном из регистров общего назначения, то при описании этой переменной вы

должны использовать директиву `register`. Вот несколько **примеров** таких описаний:

```
register int alpha;              // Определение регистровой переменной alpha
register unsigned int beta       // Определение регистровой переменной beta
register int gamma @10;          // Определение регистровой переменной gamma
                                 // с конкретным указанием регистра (R10),
                                 // где она должна храниться
```

Второй тип памяти, где могут храниться переменные — это **энергонезависимая память данных (EEPROM)**. Для размещения переменных в этой памяти используется директива `eeprom`. Причем в EEPROM могут храниться как переменные, так и массивы. Команда описания в этом случае будет выглядеть следующим образом:

```
eeprom char beta;        // Определение переменной beta с размещением в EEPROM
eeprom long array1[5]; // Определение массива array1 в EEPROM
eeprom char string[]="Hello" // Определение текстового массива
```

При размещении массива в EEPROM необходимо учитывать, что указатель массива всегда должен иметь не менее 16 битов. То есть иметь тип `int` либо `unsigned int`.

Еще один тип памяти, где могут храниться данные, — это **программная память.** Но в этом случае мы уже не сможем в программе изменять значения переменных и элементов массива. Эти значения определяются один раз и только при инициализации. Для указания того факта, что массив или переменная должны храниться в программной памяти, используется директива `flesh`.

В качестве **примера** описания массива в программной памяти обратимся к **строке 2** нашей программы (**листинг 1.16**). Массив или переменная с индексом `flesh` могут быть только глобальными. Поэтому их описание всегда располагается в начале программы перед описанием всех функций.

Листинг 1.16

```
   /*************************************************
   Project : Prog 8
   Version : 1
   Date    : 31.01.2006
   Author  : Belov
   Comments: Сигнализатор «Семь нот»

   Chip type          : ATtiny2313
   Clock frequency    : 4,000000 MHz
   *************************************************/
1  #include <tiny2313.h>

   // Объявление и инициализация массива коэффициентов деления
2  flash unsigned int tabkd[7] = {4748,4480,4228,3992,3768,3556,3356};

3  void main(void)
   {
4  unsigned char count;  // Определяем переменную count
5  unsigned char temp;   // Определяем переменную temp

6  PORTB=0x00;    // Инициализация порта PB
7  DDRB=0x08;

8  PORTD=0x7F;    // Инициализация порта PD
9  DDRD=0x00;

10 ACSR=0x80;   // Инициализация (отключение) компаратора

11 TCCR1A=0x00;    // Инициализация таймера счетчика T1
12 TCCR1B=0x09;

13 while (1)
       {
14 m1:        temp=PIND;
15            for (count=0; count<7; count++)  // Цикл сканирования датчиков
                 {
16                 if ((temp&1)==0) goto m2;    // Проверка младшего бита переменной temp
17                 temp >>= 1;                  // Сдвиг содержимого temp
                 }
18            TCCR1A=0x00;         // Выключение звука
19            goto m1;             // Переход на начало

20 m2:        OCR1A=tabkd[count];  // Запись коэффициента деления таймера.
21            TCCR1A=0x40;         // Включение звука
       };
   }
```

Описание программы (листинг 1.16)

Программа содержит только одну главную функцию `main`. Перед началом этой функции происходит определение и инициализация массива (**строка 2**). Этот массив предназначен для хранения коэффициентов деления и является аналогом таблицы коэффициентов деления в программе на Ассемблере.

При инициализации элементам массива присваиваются уже знакомые нам значения. Те же значения мы помещали в таблицу коэффициентов.

Главная функция `main` программы занимает **строки 4—21**. В **строках 4** и **5** создаются две переменные:

- переменная `count`, которая будет использоваться для определения кода нажатого датчика;
- перéменная `temp`, предназначенная для вспомогательных целей.

Строки 6—12, занимает **модуль инициализации**. Здесь настраиваются порты ввода вывода (**строки 6—9**), компаратор (**строка 10**) и таймер T1 (**строки 11, 12**). При настройке в управляющие регистры микроконтроллера записываются те же самые коды, которые мы записывали в те же регистры в программе на Ассемблере (см. **листинг 1.15**).

Основной цикл программы занимает **строки 13—21**. В **строке 14** программа читает содержимое порта PD и помещает прочитанный байт в переменную `temp`. Этот байт, как вы понимàете, содержит информацию о состоянии датчиков.

В **строках 15—17** находится **цикл сканирования датчиков**. В качестве параметра цикла используется переменная `count`. Цикл выполняется, пока значение переменной `count` меньше семи. В теле цикла происходит проверка младшего бита переменной `temp` (**строка 16**).

Для проверки значения бита используется выражение `temp&1`. Оператор «&» выполняет операцию побитового «И» между значением переменной `temp` и числом 1 (0x01). При этом все старшие биты обнуляются, и результат выражения становится равным значению младшего бита переменной `temp`.

Оператор `if` в **строке 16** проверяет результат выражения. Если он равен нулю (контакты очередного датчика замкнуты), управление передается по метке m2, и цикл прекращается досрочно. В противном случае (контакты датчика разомкнуты) цикл выполняется дальше.

В **строке 17** производится **логический сдвиг** содержимого переменной `temp`. А затем цикл сканирования начинается сначала. Таким образом, за семь проходов цикла проверяются все семь датчиков.

Если контакты всех датчиков окажутся незамкнутыми, то цикл `for` (**строки 15—17**) завершается естественным образом, а управление переходит к **строке 18**. Здесь происходит выключение звука: регистру TCCR1A присваивается нулевое значение.

Затем в **строке 19** управление передается в начало процедуры. Для передачи управления используется оператор безусловного перехода.

Если цикл завершился досрочно переходом к **строке 20** (по метке m2), начинается **процесс формирования звука.** Так как процесс сканирования датчиков закончился досрочно, в переменной `count` находится

номер датчика, контакты которого оказались замкнутыми. Теперь нам остается лишь извлечь коэффициент деления, соответствующий этому номеру датчика, и записать его в регистр совпадения таймера.

На СИ это делается элементарно. Регистру OPCR1A просто присваивается значение соответствующего элемента массива `tabkd` (**строка 20**). В **строке 21** производится включение звука. Как и в программе на Ассемблере, звук включается путем записи в регистр TCCR1A кода 0x40. На этом **завершается тело основного цикла программы**. Так как основной цикл программы выполнен в виде бесконечного цикла, то после завершения его последней команды весь цикл повторяется сначала.

1.10. Музыкальная шкатулка

Постановка задачи

Все предыдущие примеры не имели практического значения. Их можно рассматривать как простейшие задачи для тренировки. Надеюсь, к данному моменту вы достигли такого уровня знаний, что вам по плечу настоящая программа, имеющая практическое значение.

Итак, попробуем создать музыкальное устройство, автоматически воспроизводящее разные мелодии. В предыдущем примере мы научились издавать различные музыкальные звуки. Теперь нужно заставить микроконтроллер составлять из этих звуков мелодии. Причем эти мелодии должны быть жестко записаны в память микроконтроллера.

Сформулируем задачу следующим образом:

«Разработать устройство, предназначенное для воспроизведения простых одноголосых мелодий, записанных в память программ на этапе программирования. Устройство должно иметь семь управляющих кнопок. Каждой из кнопок должна соответствовать своя мелодия. Мелодия воспроизводится при нажатии и удержании кнопки. При отпускании всех кнопок воспроизведение мелодий прекращается».

Схема

Неслучайно в условии задачи заложено именно семь кнопок и одноголосые мелодии. Это позволяет использовать для новой задачи

схему из предыдущего примера (**рис. 1.14**). Представим, что контакты, подключенные к порту PD, — это не датчики, а кнопки управления. Каждая кнопка будет включать одну из семи заложенных в программу мелодий. Воспроизведение мелодий будет происходить при помощи звуковой части схемы (R1, VT1, VF1).

Алгоритм

Для начала нам нужно придумать, как мы будем хранить мелодии в памяти. Для того, чтобы в памяти можно было что-либо хранить, нужно сначала это что-то каким-либо способом закодировать. Любая мелодия состоит из нот. Каждая нота имеет свой тон (частоту) и длительность звучания. Для того, чтобы закодировать тон ноты, можно просто все ноты пронумеровать по порядку. Удобнее нумеровать, начиная с самого низкого тона. На клавиатуре клавишного инструмента это будет слева направо.

Известно, что весь музыкальный ряд делится на **октавы**. Если вы думаете, что в каждой октаве семь нот, то вы плохо знаете физические основы музыкального ряда. На самом деле в современном музыкальном ряду каждая октава делится на 12 нот. Семь основных нот (белые клавиши) и пять дополнительных (черные клавиши).

Деление на основные и дополнительные ноты сложилось исторически. В настоящее время используется музыкальный строй, в котором все 12 нот одной октавы равнозначны. Частоты любых двух соседних нот отличаются друг от друга в одинаковое количество раз. При этом частоты одноименных нот в двух соседних октавах отличаются ровно в два раза. Более подробно об этом вы можете прочитать в **[5]**.

Для нас же важно то, что коды всем этим нотам мы должны присваивать в порядке возрастания частоты. И начнем мы с ноты «До» первой октавы. Для музыкальной шкатулки более низкие ноты не нужны. В **табл. 1.3** показаны коды для всей первой октавы. Следующая, вторая октава продолжает первую и по кодировке, и по набору частот. Так нота «До» второй октавы будет иметь код 13, а частоту $f_{12}=f_0\times2$. А нота «Ре» второй октавы будет иметь код 14 и частоту $f_{13}=f_1\times2$. И так далее.

Таблица 1.3. *Кодировка нот первой октавы*

Код	Нота	Частота	Код	Нота	Частота
1	До	f_0	7	Фа#	$f_6=f_5/K$
2	До#	$f_1=f_0/K$	8	Соль	$f_7=f_6/K$
3	Ре	$f_2=f_1/K$	9	Соль#	$f_8=f_7/K$
4	Ре#	$f_3=f_2/K$	10	Ля	$f_9=f_8/K$
5	Ми	$f_4=f_3/K$	11	Ля#	$f_{10}=f_9/K$
6	Фа	$f_5=f_4/K$	12	Си	$f_{11}=f_{10}/K$

Для справки: $K = \sqrt[12]{2}$

Музыкальная длительность тоже легко кодируется. В музыке применяют не произвольную длительность, а длительность, выраженную долями от целой (см. **табл. 1.4**). В зависимости от темпа реальная длительность целой ноты меняется. Для сохранения мелодии необходимо соблюдать лишь соотношения между длительностями. Поэтому нам необходимо закодировать лишь семь вариантов длительности. Присвоим им коды от 0 до 6. Например так, как это показано в графе «Код» **табл. 1.4**. Назначение графы «Коэффициент деления» мы пока опустим.

Таблица 1.4. *Кодирование музыкальных длительностей*

Код	Длительность	Коэффициент деления
0	1 (целая)	64
1	1/2 (половинная)	128
2	1/4 (четверть)	256
3	1/8 (восьмая)	512
4	1/16 (шестнадцатая)	1024
5	1/32 (тридцать вторая)	2048
6	1/64 (шестьдесят четвертая)	4096

Кроме нот, любая мелодия обязательно содержит **музыкальные паузы**.

Определение. ***Паузы** — это промежутки времени, когда ни один звук не звучит. Длительность музыкальных пауз принимает точно такие же значения, как и длительность нот.*

В связи с этим удобно представить паузу как еще одну ноту. Ноту без звука. Такой ноте логично присвоить нулевой код.

Кодируем мелодии

Для экономии памяти удобнее каждую **ноту** кодировать одним байтом. Договоримся, что три старших бита мы будем использовать для кодирования длительности ноты, а оставшиеся пять битов — для кодирования ее тона. Пятью битами можно закодировать до 32 разных нот, что вполне хватит для музыкальной шкатулки.

Итак, если использовать приведенный выше способ кодирования, то код ноты ля первой октавы длительностью 1/4 в двоичном виде будет равен.

01001001 = 73

Код ноты (9)

Код длительности (2)

Теперь мы можем приступать к **кодированию мелодий**. Для того, чтобы закодировать мелодию, нам нужна ее **нотная запись**. Используя нотную запись, мы должны присвоить каждой ноте и каждой музыкальной паузе свой **код**.

Цепочка таких кодов и будет представлять собой закодированную мелодию. По условиям задачи наша музыкальная шкатулка должна уметь воспроизводить семь разных мелодий. Коды всех семи мелодий мы разместим в программной памяти микроконтроллера.

Как определить **конец каждой мелодии**? Для того, чтобы компьютер знал, где заканчивается каждая мелодия, используем код 255 в качестве признака конца.

Теперь нам нужно придумать, как микроконтроллер будет находить **начало каждой мелодии.** Все мелодии имеют разную длину, а в памяти они будут записаны одна за другой. Поэтому адрес начала каждой мелодии зависит от длины всех предыдущих. Удобнее всего просто по факту определить адрес начала каждой мелодии и поместить все семь адресов в специальную таблицу.

Кроме этой таблицы нам еще понадобится таблица коэффициентов деления для всех 32 нот и таблица, хранящая константы задержки для всех используемых нами музыкальных длительностей.

Алгоритм работы музыкальной шкатулки

Теперь мы можем сформулировать алгоритм работы музыкальной шкатулки.

1. Просканировать клавиатуру и определить номер самой первой нажатой клавиши.
2. Извлечь из таблицы начал мелодий значение элемента, номер которого соответствует только что определенному номеру нажатой клавиши. Это значение будет равно адресу в программной памяти, где начинается нужная нам мелодия.
3. Начать цикл воспроизведения мелодии. Для этого поочередно извлекать коды нот из памяти, начиная с адреса, который мы определили в п.2 алгоритма.
4. Каждый код ноты разложить на код тона и код длительности.
5. Если код тона равен нулю, отключить звук и перейти к формированию задержки (к п. 9 настоящего алгоритма).
6. Если код тона не равен нулю, извлечь из таблицы коэффициентов деления значение элемента с номером, равным коду тона.
7. Записать коэффициент деления, который мы нашли в пункте 6 настоящего алгоритма, в регистр совпадения таймера T1.
8. Включить звук (подключить вывод OC1A к выходу таймера T1).
9. Извлечь из таблицы длительностей задержки значение элемента с номером, равным коду длительности.
10. Сформировать паузу с использованием константы задержки, которую мы нашли в пункте 9 настоящего алгоритма.
11. По окончании паузы выключить звук (отключить OC1A от выхода таймера).
12. Повторять цикл (пункты 4—11 настоящего алгоритма) до тех пор, пока нажата соответствующая кнопка.
13. Если очередной код ноты окажется равным 255, перейти к началу текущей мелодии, то есть вернуться к п. 3 настоящего алгоритма.

Программа на Ассемблере

Возможный вариант программы на языке Ассемблер приведен в **листинге 1.17**. В программе используются следующие новые для нас операторы.

andi

Логическое «И» содержимого РОН и константы. Команда имеет следующий формат:

```
andi Rd, K,
```

где Rd — имя регистра общего назначения, а K — некая числовая константа.

Команда andi выполняет операцию побитового «И» между содержимым регистра и константой. Результат помещается в регистр Rd.

Команда andi часто используется как *операция наложения маски*. Что в данном случае означает понятие «маска»? Для того, чтобы это объяснить, я хочу немного отвлечься и обратиться к детективному жанру. Если точнее, я предлагаю вспомнить один из методов шифрования различных посланий.

Помните заставку к фильму о Шерлоке Холмсе? На лист бумаги, на котором в хаотическом на первый взгляд порядке записаны случайные знаки, накладывается второй лист, в котором в определенных местах проделаны дырочки. При совмещении этих двух листов лишние знаки оказываются невидимыми, а сквозь дырочки мы видим символы, составляющие зашифрованное сообщение.

Что-то подобное происходит при наложении маски с использованием двух двоичных чисел. Одно из этих чисел является маской для второго числа. При наложении маски лишние биты обнуляются, а остаются лишь те, которые несут нужную для нас информацию.

В нашей программе наложение маски используется для того, чтобы из **кода ноты** выделить **код тона** или **код длительности**. Код тона занимает пять младших разрядов общего кода. Для выделения кода тона нужно произвести операцию побитового «И» между кодом ноты и маской. Маска в данном случае должна быть равна 00011111B (или в шестнадцатиричном виде — 1FH). Выделение кода тона происходит в **строке 67** программы (см. **листинг 1.17**). В результате наложения маски три старших бита обнуляются, а пять младших остаются без изменений. Для выделения кода длительности выбирается другое значение маски (см. **строку 74**).

adiw

Шестнадцатиразрядное сложение. Позволяет прибавить к шестнадцатиразрядному числу некоторую числовую константу. Шестнадцатиразрядное значение помещается в регистровую пару. При этом может использоваться одна из следующих пар: R24:R25, R26:R27, R28:R29 и R30:R31.

Команда имеет ограничение на величину константы. Константа может иметь значение 0 до 63. Команда имеет два параметра:

- первый параметр — это имя первого из регистров регистровой пары;
- второй параметр — это прибавляемая константа.

По странной прихоти разработчиков данной версии Ассемблера, в качестве первого параметра нельзя использовать имена регистровых пар X, Y и Z, а также их половинок (например, XL или YH).

В **строке 126** нашей программы (**листинг 1.17**) к содержимому регистровой пары R30:R31 прибавляется единица. По сути, в данном случае мы добавляем константу к содержимому регистровой пары Z.

cp

Сравнение двух РОН. Команда имеет два параметра. Оба параметра — это имена регистров общего назначения. Команда не изменяет содержимого самих регистров. Она просто производит их сравнение. По результатам сравнения устанавливаются флаги в регистре SREG. После команды сравнения обычно применяется одна из команд условного перехода.

rol

Циклический сдвиг содержимого регистра влево через признак переноса. Это еще один оператор сдвига. В данном случае сдвиг битов производится по кругу, как показано на **рис. 1.15**. В этот круг включена ячейка признака переноса C.

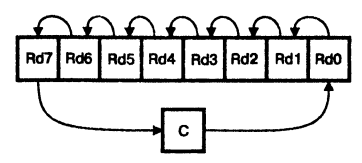


Рис. 1.15. *Схема выполнения операции циклического сдвига влево*

.db

Директива описания данных. Эта директива так же, как и уже известная нам директива dw предназначена для описания данных, помещаемых в память программ или в EEPROM. Директива db, в отличие от dw, описывает не двухбайтовые слова данных, а отдельные байты. Значения, которые нужно поместить в память, записываются справа после оператора db через запятую. Каждое такое значение не должно превышать 255 (максимальное число, которое можно записать при помощи одного байта).

Если данных очень много, для их записи можно применять несколько операторов db, расположенных непосредственно друг за другом. Однако при этом нужно учитывать один важный момент. Сколько бы значений ни было записано в правой части директивы db, она всегда записывает четное их количество. Это связано с указателем текущего адреса в сегменте cseg.

Указатель работает с основной адресацией и должен переместиться на целое количество ячеек памяти в соответствии с этой адресацией. А в этой адресации каждая ячейка вмещает в себя два байта данных. Поэтому в том случае, если программист все же укажет нечетное количество параметров, то директива db в конце припишет еще один, который будет равен нулю. Поэтому, если вам необходимо записать в память большое количество значений и вы собираетесь разбить все параметры на несколько строк и записать каждую **строку** при помощи директивы db, то во всех **строках**, кроме последней, вы должны оставить четное количество параметров.

Листинг 1.17

```
;###############################################
;##                Пример 9                   ##
;##          Музыкальная шкатулка             ##
;###############################################

;------------------------------ Псевдокоманды управления
.include "tn2313def.inc"                ; Присоединение файла описаний
.list                                   ; Включение листинга

.def      loop1 = R0                    ; Три ячейки для процедуры задержки
.def      loop2 = R1
.def      loop3 = R21
.def      temp = R16                    ; Вспомогательный регистр
.def      temp1 = R17                   ; Второй вспомогательный регистр
.def      count = R17                   ; Определение регистра счетчика опроса клавиш
.def      fnota = R19                   ; Частота текущей ноты
.def      dnota = R20                   ; Длительность текущей ноты
```

```
;------------------------ Начало программного кода

            .cseg                       ; Выбор сегмента программного кода
             org       0                ; Установка текущего адреса на ноль

start:      rjmp       init             ; Переход на начало программы
            reti                        ; Внешнее прерывание 0
            reti                        ; Внешнее прерывание 1
            reti                        ; Таймер/счетчик 1, захват
            reti                        ; Таймер/счетчик 1, совпадение, канал A
            reti                        ; Таймер/счетчик 1, прерывание по переполнению
            reti                        ; Таймер/счетчик 0, прерывание по переполнению
            reti                        ; Прерывание UART прием завершен
            reti                        ; Прерывание UART регистр данных пуст
            reti                        ; Прерывание UART передача завершена
            reti                        ; Прерывание по компаратору
            reti                        ; Прерывание по изменению на любом контакте
            reti                        ; Таймер/счетчик 1. Совпадение, канал B
            reti                        ; Таймер/счетчик 0. Совпадение, канал B
            reti                        ; Таймер/счетчик 0. Совпадение, канал A
            reti                        ; USI готовность к старту
            reti                        ; USI Переполнение
            reti                        ; EEPROM Готовность
            reti                        ; Переполнение охранного таймера

;*****************************************************
;*              Модуль инициализации
;*****************************************************
init:
;------------------------- Инициализация стека

            ldi        temp, RAMEND     ; Инициализация стека
            out        SPL, temp

;------------------------- Инициализация портов В/В

            ldi        temp, 0x08       ; Инициализация порта PB
            out        PORTB, temp
            out        DDRB, temp

            ldi        temp, 0x7F       ; Инициализация порта PD
            out        PORTD, temp
            ldi        temp, 0x00
            out        DDRD, temp

;-------------------------- Инициализация (выключение) компаратора

            ldi        temp, 0x80
            out        ACSR, temp

;-------------------------- Инициализация таймера T1

            ldi        temp, 0x09       ; Включаем режим CTC
            out        TCCR1B, temp
m1:         ldi        temp, 0x00       ; Выключаем звук
            out        TCCR1A, temp


;*****************************************************
;*            Начало основной программы          *
;*****************************************************
main:
;------------------------- Вычисление номера нажатой кнопки

            clr        count            ; Обнуление счетчика опроса клавиш
            in         temp, PIND       ; Чтение порта D
m2:         lsr        temp             ; Сдвигаем входной байт
            brcc       m3               ; Если текущий разряд был равен 0
            inc        count            ; Увеличиваем показание счетчика
            cpi        count, 7         ; Сравнение (7 - конец сканирования)
            brne       m2               ; Если не конец, продолжить
            rjmp       m1               , Если не одна клавиша не нажата

;------------------------- Выбор мелодии

m3:         mov        YL, count        ; Вычисляем адрес, где
```

```
            ldi     ZL, low(tabm*2)     ; хранится начало мелодии
            ldi     ZH, high(tabm*2)
            rcall   addw                ; К подпрограмме 16-разрядного сложения

            lpm     XL, Z+              ; Извлекаем адреса из таблицы
            lpm     XH, Z               ; и помещаем в X

;-------------------------- Воспроизведение мелодии

m4:         mov     ZH, XH              ; Записываем в Z начало мелодии
            mov     ZL, XL

m5:         in      temp, PIND          ; Читаем содержимое порт D
            cpi     temp, 0x7F          ; Проверяем на равенство 7FH
            breq    m1                  ; Если равно (кнопки отпущены) в начало

            lpm     temp, Z             ; Извлекаем код ноты
            cpi     temp, 0xFF          ; Проверяем, не конец ли мелодии
            breq    m4                  ; Если конец, начинаем мелодию сначала

            andi    temp, 0x1F          ; Выделяем код тона из кода ноты
            mov     fnota, temp         ; Записываем в регистр кода тона
            lpm     temp, Z+            ; Еще раз берем код ноты
            rol     temp                ; Производим четырехкратный сдвиг кода ноты
            rol     temp
            rol     temp
            rol     temp
            andi    temp, 0x07          ; Выделяем код длительности
            mov     dnota, temp         ; Помещаем ее в регистр длительности

            rcall   nota                ; К подпрограмме воспроизведения ноты

            rjmp    m5                  ; В начало цикла (следующая нота)


;****************************************************
;*          Вспомогательные подпрограммы            *
;****************************************************

; --------------------- Подпрограмма 16-ти разрядного сложения
addw:       push    YH

            lsl     YL                  ; Умножение первого слагаемого на 2
            ldi     YH, 0               ; Второй байт первого слагаемого = 0
            add     ZL, YL              ; Складываем два слагаемых
            adc     ZH, YH

            pop     YH
            ret

;--------------------- Подпрограмма исполнения одной ноты
nota:       push    ZH
            push    ZL
            push    YL
            push    temp

            cpi     fnota, 0x00         ; Проверка, не пауза ли
            breq    nt1                 ; Если пауза, переходим сразу к задержке

            mov     YL, fnota           ; Вычисляем адрес, где хранится
            ldi     ZL, low(tabkd*2)    ; коэффициент деления для текущей ноты
            ldi     ZH, high(tabkd*2)
            rcall   addw                ; К подпрограмме 16-разрядного сложения

            lpm     temp, Z+            ; Извлекаем мл. разряд КД для текущей ноты
            lpm     temp1, Z            ; Извлекаем ст. разряд КД для текущей ноты
            out     OCR1AH, temp1       ; Записать в старш. часть регистра совпадения
            out     OCR1AL, temp        ; Записать в младш часть регистра совпадения

            ldi     temp, 0x40          ; Включить звук
            out     TCCR1A, temp

nt1:        rcall   wait                ; К подпрограмме задержки
```

```
                ldi     temp, 0x00          ; Выключить звук
                out     TCCR1A, temp

                ldi     dnota,0             ; Сбрасываем задержку для паузы между нотами
                rcall   wait                ; Пауза между нотами

                pop     temp                ; Завершение подпрограммы
                pop     YL
                pop     ZL
                pop     ZH
                ret

;---------------------- Подпрограмма формирования задержки
wait:           push    ZH
                push    ZL
                push    YH
                push    YL

                mov     YL, dnota           ; Вычисляем адрес, где хранится
                ldi     ZL, low(tabz*2)     ; нужный коэффициент задержки
                ldi     ZH, high(tabz*2)
                rcall   addw                ; К подпрограмме 16-разрядного сложения

                lpm     YL, Z+              ; Читаем первый байт коэффициента задержки
                lpm     YH, Z               ; Читаем второй байт коэффициента задержки

                clr     ZL                  ; Обнуляем регистровую пару Z
                clr     ZH

                ; Цикл задержки
w1:             ldi     loop,255            ; Пустой внутренний цикл
w2:             dec     loop
                brne    w2
                adiw    R30, 1              ; Увеличение регистровой пары Z на единицу
                cp      YL, ZL              ; Проверка младшего разряда
                brne    w1
                cp      YH, ZH              ; Проверка старшего разряда
                brne    w1

                pop     YL                  ; Завершение подпрограммы
                pop     YH
                pop     ZL
                pop     ZH
                ret

;****************************************************
;*          Таблица длительности задержек           *
;****************************************************

tabz:           .dw     128,256,512,1024,2048,4096, 8192

;****************************************************
;*          Таблица коэффициентов деления           *
;****************************************************

tabkd:          .dw     0
                .dw     4748,4480,4228,3992,3768,3556,3356,3168,2990,2822,2664, 2514
                .dw     2374,2240,2114,1996,1884,1778,1678,1584,1495,1411,1332, 1257
                .dw     1187,1120,1057, 998, 942, 889, 839, 792

;****************************************************
;*          Таблица начал всех мелодий              *
;****************************************************

tabm:           .dw     mel1*2,mel2*2,mel3*2,mel4*2
                .dw     mel5*2,mel6*2,mel7*2
```

```
;*****************************************************
;*                  Таблица мелодий                  *
;*****************************************************
;                 В траве сидел кузнечик
mel1:             .db      109,104,109,104,109,108,108, 96,108,104
                  .db      108,104,108,109,109, 96,109,104,109,104
                  .db      109,108,108, 96,108,104,108,104,108,141
                  .db       96,109,111, 79, 79,111,111,112, 80, 80
                  .db      112,112,112,111,109,108,109,109, 96,109
                  .db      111, 79, 79,111,111,112, 80, 80,112,112
                  .db      112,111,109,108,141,128, 96,255
;                 Песенка крокодила Гены
mel2:             .db      109,110,141,102,104,105,102,109,110,141
                  .db      104,105,107,104,109,110,141,104,105,139
                  .db      109,110,173, 96,114,115,146,109,110,112
                  .db      109,114,115,146,107,109,110,114,112,110
                  .db      146,109,105,136,107,105,134,128,128,102
                  .db      105,137,136,128,104,107,139,137,128,105
                  .db      109,141,139,128,110,109,176,112,108,109
                  .db      112,144,142,128,107,110,142,141,128,105
                  .db      109,139,128,173,134,128,128,109,112,144
                  .db      142,128,107,110,142,141,128,105,109,139
                  .db      128,173,146,128, 96,255
;                 В лесу родилась елочка
mel3:             .db      132,141,141,139,141,137,132,132,132,141
                  .db      141,142,139,176,128,144,146,146,154,154
                  .db      153,151,149,144,153,153,151,153,181,128
                  .db     96,255
;                 Happy births to you
mel4:             .db      107,107,141,139,144,143,128,107,107,141
                  .db      139,146,144,128,107,107,151,148,146,112
                  .db      111,149,117,117,148,144,146,144,128,255
;                 С чего начинается родина
mel5:             .db       99,175,109,107,106,102, 99,144,111,175
                  .db       96, 99,107,107,107,107,102,104,170, 96
                  .db     99,109,109,109,109,107,106,143,109,141
                  .db     99,109,109,109,109,104,106,171, 96, 99
                  .db      111,109,107,106,102, 99,144,111,143,104
                  .db      114,114,114,114,109,111,176, 96,104,116
                  .db      112,109,107,106, 64, 73,143,107,131, 99
                  .db      144, 80, 80,112,111, 64, 75,173,128,255
;                 Песня из кинофильма "Веселые ребята"
mel6:             .db      105,109,112,149,116, 64, 80,148,114, 64
                  .db       78,146,112, 96,105,105,109,144,111, 64
                  .db       80,145,112, 64, 81,178, 96,117,117,117
                  .db      149,116, 64, 82,146,112, 64, 79,146,144
                  .db       96,105,105,107,141,108,109,112,110,102
                  .db      104,137,128, 96,105,105,105,137,102, 64
                  .db       73,142,105,107,109, 64, 75,137, 96,105
                  .db      105,105,137,102,105,142,112, 64, 82,180
                  .db       96,116,116,116,148,114,112,142,109, 64
                  .db       78,146,144, 96,105,105,107,141,108,109
                  .db      112,110,102,104,169, 96, 96,255
;                 Улыбка
mel7:             .db      107,104,141,139,102,105,104,102,164,128
                  .db      104,107,109,109,109,111,114,112,111,109
                  .db      144,139,128,109,111,144, 96,111,109,104
                  .db      107,105,173,128,111,109,112,107,111,109
                  .db      109,107,102,104,134,132,128,100,103,107
                  .db      107,107,107,139,112,100,103,102,102,102
                  .db      134,102,103,107,105,107,108,108,108,108
                  .db      107,105,107,108,144,142,128,112,107,110
```

```
195          .db       140,112,105,108,107,107,107,105,140,139
196          .db       139,112,103,102,103,105,108,107,105,103
197          .db       128,112,107,110,108,108,108,140,112,105
198          .db       108,107,107,107,139,112,103,102,103,105
199          .db       108,107,105,103,105,139,132,128, 96, 96
200          .db        96,255
```

Описание программы (листинг 1.17)

Благодаря тому, что принципиальная схема, назначение выводов и режимы работы портов с предыдущего раза остались без изменений, то и новая наша программа во многом похожа на предыдущую.

Главное **отличие** новой программы — наличие не одной, а нескольких таблиц в памяти программ. Все таблицы помещаются в конце программы. Для описания данных, размещаемых в этих таблицах, применяются как директивы `db`, так и директивы `dw`.

Первая таблица содержит коэффициенты задержки для формирования всех вариантов музыкальной длительности. Таблица начинается с адреса, соответствующего метке `tabz`. Вся таблица занимает одну **строку** программы (**строка 136**). Так как в нашей программе мы будем применять лишь семь вариантов длительности, таблица имеет 7 элементов. Каждый элемент записывается в память как двухбайтовое слово.

В **строках 137—140** описывается **таблица коэффициентов деления для всех нот**. Начало таблицы соответствует метке `tabkd`. Каждый элемент этой таблицы также имеет размер в два байта. Первый элемент таблицы равен нулю. Это неиспользуемый элемент. Ноты номер ноль у нас не существует. Ноль мы использовали для кодирования паузы.

В паузе не формируется звуковой сигнал, поэтому и коэффициент деления там не имеет смысла. Поэтому значение нулевого элемента массива несущественно. Описание таблицы разбито на строки. Для удобства каждая строка описывает коэффициенты деления для одной октавы. Нулевая нота выделена в отдельную строку. Последняя октава неполная, так как наша шкатулка будет использовать всего 32 ноты.

В **строках 143—200** описана **таблица мелодий**. Вернее, это не одна таблица, а семь таблиц (своя таблица для каждой из мелодий). Каждая

таблица помечена своей отдельной меткой (mel1, mel2 — mel7). Значение каждой метки — это адрес начала соответствующей мелодии. Каждое значение таблицы мелодий записывается в память в виде одного байта. Поэтому все строки, кроме последней, для каждой таблицы имеют четное число значений.

В **строках 141, 142** описана **таблица начал всех мелодий**. Начало этой таблицы отмечено меткой tabm. Таблица используется для того, чтобы программа могла найти адрес начала нужной мелодии по ее номеру. В качестве элементов массива выступают удвоенные значения меток mel1, mel2 — mel7. Применение удвоенных значений обусловлено необходимостью перевода адресов из основной адресации в альтернативную. При трансляции программы вместо меток в память будут записаны конкретные адреса.

Процедура вычисления адреса

Большое количество таблиц в нашей программе заставляет позаботиться о процедуре вычисления адреса. Как нам известно из предыдущей программы, для извлечения элемента из таблицы нам необходимо произвести вычисление его адреса. В новой программе подобные вычисления нам придется выполнять для каждой таблицы.

Однотипные вычисления удобно оформить в виде подпрограммы. Эта подпрограмма занимает **строки 78—84**. Вызов подпрограммы производится по имени addw. Подпрограмма получает номер элемента таблицы и адрес ее начала. Номер элемента передается в подпрограмму при помощи регистра YL, а адрес — через регистровую пару Z.

Используя эти данные, подпрограмма вычисляет **адрес нужного элемента**. Для этого она сначала удваивает номер элемента (**строка 79**). Затем дополняет полученное значение до шестнадцатиразрядного путем записи в YH нулевого байта (**строка 80**). И, наконец, производит сложение двух шестнадцатиразрядных величин, находящихся к этому моменту в регистровых парах Y и Z (**строки 81, 82**). Результат вычислений при этом попадает в регистровую пару Z. Назначение команд push и pop (**строки 78, 83**), по-видимому, уже объяснять не нужно.

Текст программы «шаг за шагом»

Теперь вернемся к самому началу и рассмотрим текст программы по порядку. **Начало программы** практически полностью соответствует предыдущему примеру (см. **листинг 1.15**). Небольшое отличие лишь в модуле описания переменных (в новой программе это **строки 3—10**). Теперь там описывается гораздо больше переменных (рабочих регистров).

Название и назначение новых переменных прекрасно видны из текста программы. Без изменений остался модуль переопределения векторов прерываний (**строки 13—31**) и модуль команд инициализации (**строки 32—44**). Не изменилась даже процедура сканирования управляющих кнопок (**строки 45—52**), которая так же, как и в предыдущем примере, определяет номер первого из входов, у которого оказались замкнуты контакты. На этом сходство двух программ заканчивается. Начиная со **строки 53**, мы видим абсолютно новую программу. Рассмотрим ее подробнее.

Особенности программы

Итак, процедура, расположенная в **строках 45—52** программы, просканировала клавиатуру и нашла код первой из нажатых кнопок. Искомый код, если вы не забыли, находится в регистре `count`. Затем управление переходит к **строке 53**. С этого места начинается процедура выбора мелодии (**строки 53—58**). Суть процедуры — прочитать из таблицы `tabm` значение адреса начала этой мелодии. То есть прочитать элемент таблицы, номер которого равен коду нажатой клавиши.

Прежде чем прочитать элемент, необходимо **найти его адрес**. Для вычисления адреса используем подпрограмму `addw`. Перед тем, как вызвать подпрограмму, подготовим все данные. Номер нажатой клавиши помещаем в регистр YL (**строка 53**). Адрес начала таблицы записываем в регистровую пару Z (**строки 54, 55**). И лишь затем в **строке 56** вызывается подпрограмма `addw`.

После выхода из подпрограммы в регистровой паре Z находится результат вычислений — адрес нужного нам элемента таблицы `tabm`. Следующие две команды (**строки 57** и **58**) извлекают тот элемент (адрес начала мелодии) и помещают его в регистровую пару X. Там этот адрес будет храниться все время, пока воспроизводится именно эта мелодия.

Следующий этап — воспроизведение мелодии. Воспроизведением мелодии занимается процедура, расположенная в **строках 59—77**. Для последовательного воспроизведения нот нам понадобится указатель текущей ноты. В качестве указателя текущей ноты используется регистровая пара Z. В самом начале процедуры воспроизведения мелодии в регистровую пару Z помещается адрес начала мелодии их регистровой пары X (**строки 59, 60**).

Затем начинается цикл воспроизведения (**строки 61—77**). В этом цикле программа извлекает код ноты по адресу, на который указывает наш указатель, выделяет из кода ноты код тона и код длительности, воспроизводит ноту, а затем увеличивает значение указателя на единицу. Затем весь цикл повторяется.

Этот процесс происходит до тех пор, пока код очередной ноты не окажется равным 255 (метка конца мелодии). Прочитав этот код, программа передает управление на **строку 62**, где в регистр Z снова записывается адрес начала мелодии. Воспроизведение мелодии начнется сначала. Этот процесс должен прерваться лишь в одном случае — при отпускании управляющей кнопки.

Для проверки состояния кнопок в цикл воспроизведения мелодии включена специальная процедура (**строки 61—63**). Процедура упрощенно проверяет состояние сразу всех кнопок. Она считывает содержимое порта PD (**строка 61**) и сравнивает его с кодом 0x7F (**строка 62**). Прочитанное из порта значение может быть равно 0x7F только в одном случае — если все кнопки отпущенны. Если хотя бы одна кнопка нажата, то при чтении порта мы получим другое значение.

Проверкой вышеописанного условия занимается оператор `breq` в **строке 63**. Если все кнопки оказались отпущены, этот оператор завершает цикл воспроизведения мелодии и передает управление на метку `m1`, то есть на самое начало основного цикла программы. Там происходит выключение звука, а затем новое сканирование клавиатуры.

Если хотя бы одна кнопка окажется нажатой, то цикл воспроизведения звука продолжается дальше, и управление переходит к **строке 64**, где происходит извлечение кода ноты. Так как адрес этой ноты находится в регистровой паре Z (указатель текущей ноты), то для извлечения ноты просто используется команда `lpm`.

В **строке 65** происходит проверка признака конца мелодии. Только что прочитанный код ноты сравнивается с кодом 0xFF. Оператор `breq` в **строке 66** передает управление по метке `m4`, если мелодия действительно закончилась (условие выполняется). Если код ноты

не равен 0xFF, перехода не происходит, и управление переходит к **строке 67**.

В **строках** 67—75 происходит обработка кода ноты. То есть из кода ноты выделяется код тона и код длительности. Сначала на код ноты накладывается маска, которая оставляет пять младших разрядов, а три старших сбрасывает (**строка 67**). Под действием маски в регистре `temp` остается код тона, который затем помещается в регистр `fnota` (**строка 68**).

Теперь нам нужно найти **код длительности ноты**. Для этого нам заново придется извлечь код ноты из памяти программ. Так как до этого момента мы не изменяли положение указателя текущей ноты, то для извлечения нет никаких препятствий. В **строке 69** мы повторно извлекаем код ноты из памяти программ. Но на этот раз значение указателя увеличивается. Теперь можно приступать к выделению кода длительности. Как вы помните, длительность кодируется тремя младшими битами кода ноты. Для выделения этих битов нам также нужно использовать **маску**. Но одной маской нам не обойтись. Нам нужно не просто выделить три старших разряда, а сделать их младшими, как это показано на **рис. 1.16**.

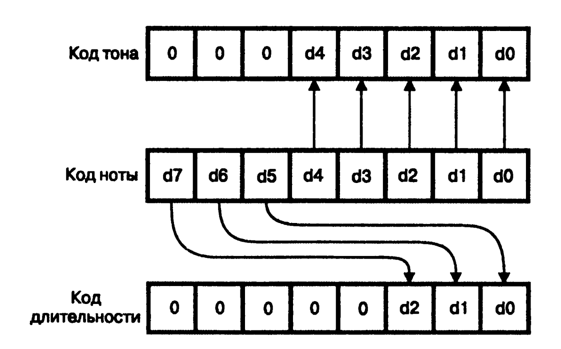


Рис. 1.16. *Разложение кода ноты*

Процедура выделения кода длительности занимает **строки 70—74**. Сначала программа производит многократный циклический сдвиг кода ноты до тех пор, пока три старших разряда не станут тремя младшими. Для сдвига используется команда `rol`. Так как сдвиг происходит через ячейку признака переноса (см. **рис. 1.15**), то нам понадобится четыре команды сдвига. Эти команды занимают в программе **строки 70—73**.

Затем в **строке 74** на полученное в результате сдвигов число накладывается маска, которая выделяет три младшие бита, а пять старших сбрасывает в ноль. Полученный таким образом код длительности записывается в регистр `dnota` (**строка 75**).

Когда **код тона** и код длительности определены, производится вызов подпрограммы воспроизведения ноты (**строка 76**). Оператор `rjmp` в **строке 77** передает управление на начало цикла воспроизведения мелодии, и цикл повторяется для следующей ноты.

Подпрограмма воспроизведения ноты занимает **строки 85—110**. Она выполняет следующие действия:

- извлекает из таблицы `tabkd` коэффициент деления, соответствующий коду ноты;
- программирует таймер и включает звук;
- затем выдерживает паузу и звук выключает.

Если код тона равен нулю (нужно воспроизвести паузу без звука), извлечение коэффициента деления и включение звука не выполняется. Подпрограмма сразу переходит к формированию паузы.

Начинается подпрограмма воспроизведения ноты с сохранения всех используемых регистров (**строки 85—88**). Затем производится проверка кода ноты на равенство нулю (**строка 89**). Если код ноты равен нулю, то оператор `breq` в **строке 90** передает управление по метке `nt1`, то есть к строке, где происходит вызов процедуры формирования задержки.

Если код ноты не равен нулю, то программа приступает к извлечению коэффициента деления. Для вычисления адреса элемента таблицы `tabkd`, где находится этот коэффициент, снова используется подпрограмма `addw`.

Код тона помещается в регистр YL (**строка 91**), а адрес начала таблицы — в регистровую пару Z (**строки 92, 93**). Вызов подпрограммы `addw` производится в **строке 94**. В регистровой паре Z подпрограмма возвращает адрес элемента таблицы, где находится нужный нам коэффициент деления. В **строках 95, 96** из таблицы извлекается этот коэффициент. А в **строках 97, 98** он помещается в регистр совпадения таймера. В **строках 99, 100** включается звук.

В **строке 104** вызывается специальная подпрограмма, предназначенная для формирования задержки. Подпрограмма называется `wait` и формирует задержку с переменной длительностью. Длительность задержки зависит от значения регистра `dnota`. По окончании задержки звук выключается (**строки 102, 103**).

На этом можно было бы закончить процесс воспроизведения ноты. Однако это еще не все. Для правильного звучания мелодии между двумя соседними нотами необходимо обеспечить хотя бы **небольшую паузу**. Если такой паузы не будет, ноты будут звучать слитно. Это исказит мелодию, особенно если подряд идет несколько нот с одинаковым тоном. Формирование паузы между нотами происходит в **строках 104, 105**.

Вспомогательная пауза формируется при помощи уже знакомой нам подпрограммы задержки. В **строке 104** коду паузы присваивается нулевое значение (выбирается самая минимальная пауза). Затем в **строке 105** вызывается подпрограмма `wait`. После окончания паузы остается только восстановить содержимое всех сохраненных регистров из стека (**строки 106—109**) и выйти из подпрограммы (**строка 110**).

Подрограмма формирования задержки

И последнее, что нам еще осталось рассмотреть, — это подпрограмма формирования задержки. Текст подпрограммы занимает **строки 111—135**. Как и любая другая подпрограмма, подпрограмма `wait` в начале сохраняет (**строки 111—114**), а в конце — восстанавливает (**строки 131—134**) все используемые регистры.

Рассмотрим, как работает эта подпрограмма. Сначала определяется длительность задержки. Для этого извлекается соответствующий элемент из таблицы `tabz`. Номер элемента соответствует коду задержки, находящемуся в регистре `dnota`. Извлечение значения из таблицы производится уже знакомым нам образом. Команды, реализующие вычисление адреса нужного элемента таблицы, находятся в **строках 115—118**. Затем в **строках 119** и **120** производится чтение элемента таблицы. Прочитанный код задержки помещается в регистровую пару Y.

Теперь **наша задача**: *сформировать задержку, пропорциональную содержимому регистровой пары Y*. Так как микроконтроллер Atiny2313 имеет только один шестнадцатиразрядный таймер, который уже занят формированием звука, будем формировать задержку программным путем. Ранее мы уже применяли один вариант такой подпрограммы (см. **раздел 1.5**). Но в данном случае цикл формирования задержки построен немного по-другому.

Вообще-то, способов построения подобных подпрограмм может быть бесконечное множество. Все зависит от изобретательности програм-

миста. Использованный в данном примере способ более удобен для формирования задержки переменной длительности, пропорциональной заданному коэффициенту. Главной **особенностью** нового способа является шестнадцатиразрядный параметр цикла.

Для хранения этого параметра используется регистровая пара Z. Перед началом цикла задержки в нее записывается ноль. Затем начинается цикл, на каждом проходе которого содержимое регистровой пары Z увеличивается на единицу. После каждого такого увеличения производится сравнение нового значения Z с содержимым регистровой пары Y.

Заканчивается цикл тогда, когда содержимое Z и содержимое Y окажутся равны. В результате число, записанное в регистровой паре Y, будет определять количество проходов цикла. Поэтому и время задержки, формируемое этим циклом, будет пропорционально константе задержки. Однако это время будет слишком мало для получения приемлемого темпа воспроизведения мелодий. Для того, чтобы увеличить время до нужной нам величины, внутрь главного цикла задержки помещен еще один цикл, имеющий фиксированное количество проходов.

Описанная выше процедура задержки занимает **строки 121—135**. В **строках 121, 122** производится запись нулевого значения в регистровую пару Z. Большой цикл задержки занимает **строки 123—130**. Малый внутренний цикл занимает **строки 124—125**. Для хранения параметра малого цикла используется регистр `loop`. В **строке 123** в него записывается начальное значение. **Строки 124, 125** выполняются до тех пор, пока содержимое `loop` не окажется равным нулю.

В **строке 126** содержимое регистровой пары Z увеличивается на единицу. В **строках 127—130** производится сравнение содержимого двух регистровых пар Y и Z. Сравнение производится побайтно. Сначала сравниваются младшие байты (**строка 127**). Если они не равны, оператор условного перехода в **строке 128** передает управление на начало цикла.

Если младшие байты равны, сравниваются старшие байты (**строка 129**). Если старшие байты неодинаковы, оператор `brne` в **строке 130** опять заставляет цикл начинаться с начала. И только когда оба оператора сравнения дадут положительный результат (не вызовут перехода), цикл заканчивается, и подпрограмма формирования задержки переходит к завершающей фазе (к **строкам 131—135**).

Программа на языке СИ

Возможный вариант программы на языке СИ приведен в **листинге 1.18**. Прежде чем начинать рассмотрение текста этой программы, необходимо разобраться с одним важным вопросом. В новой программе используется такой новый для нас элемент, как ссылочная (индексная) переменная.

Ссылочная переменная — это еще одна фирменная особенность классического СИ, которая широко применяется и позволяет сделать программы проще и эффективнее. В то же время никакой другой элемент не вызывает столько сложностей у начинающих программистов, как ссылочные переменные. Попробую объяснить этот вопрос как можно понятнее.

Итак, ссылочная переменная — это переменная, которая хранит указатель на другой объект. Таким объектом может быть либо другая переменная, либо элемент массива. При описании ссылочной переменной перед ее именем ставится символ * (звездочка). Вот как выглядит типичное описание ссылочной переменной:

```
int *sper;
```

В приведенном примере описывается ссылочная `sper` переменная, которая может хранить указатель на любую другую переменную или на любой элемент любого массива, но только если эта переменная или массив имеют тип `int`. Рассмотрим подробнее случай, когда **ссылочная переменная указывает на элемент массива**. Для того, чтобы поместить в ссылочную переменную указатель, указывающий на нулевой элемент массива `mass`, достаточно выполнить следующую команду:

```
sper = &mass[0];
```

Символ «&», поставленный перед переменной или массивом, — это операция определения ссылки на объект. Выражение `&mass[0]` возвращает указатель на нулевой элемент массива `mass`. Выражение `&mass[5]` возвращает указатель на пятый элемент. И так далее. Если нас интересует только начало массива, возможна и более простая запись. Допустим, ссылочной переменной `sper` нужно присвоить значение указателя на начало массива `mass` (то есть на его нулевой элемент). В этом случае можно записать

```
sper=mass;
```

Другими словами, в языке СИ имя массива является одновременно и указателем на его начало.

Допустим, переменная sper содержит указатель на нулевой элемент массива. Тогда sper+1 будет указывать на первый элемент того же массива, sper+2 на второй, и так далее. Если теперь переменной sper присвоить значение указателя — другой массив, то указанным выше способом мы получим доступ к новому массиву. Таким образом, используя одну и ту же ссылочную переменную можно обращаться к любому элементу любого массива.

Обращаться к разным элементам одного массива можно и по-другому. Можно увеличивать значение самой ссылочной переменной. **Например**, если увеличить значение переменной sper на единицу (sper=sper+1), то после этого она будет указывать уже на следующий элемент массива.

Хочу обратить ваше внимание на то, что ссылочная переменная хранит не адрес в памяти, где хранится элемент массива, а именно указатель. **Указатель** — это не совсем адрес. Если переменная имеет тип char, то она занимает в памяти один байт. Переменная типа int занимает два байта.

Указатель автоматически учитывает этот факт. При увеличении значения указателя на единицу, он всегда указывает на следующий элемент массива, независимо от того, сколько байтов в памяти занимает каждый такой элемент.

Теперь посмотрим, как используются **указатели в программе**. Допустим, у нас есть ссылочная переменная sper, которая содержит указатель на начало массива mass. Тогда вместо выражения

```
x = mass[1];
```

можно записать

```
x = *sper;
```

В данном случае символ * означает операцию обращения к содержимому объекта, на который ссылается переменная sper. Если теперь увеличить значение переменной sper (например, при помощи команды sper++), то после этого выражение x=*sper присвоит переменной x значение уже следующего элемента массива (mass[2]).

Как видите, применение ссылочных переменных позволяет обратиться к любому элементу массива альтернативным способом. Все удобство такого способа иллюстрирует как раз наш программный пример. Как вы помните, музыкальная шкатулка должна воспроизводить одну из семи мелодий. Разумеется, для каждой из мелодий мы создадим свой отдельный массив, куда поместим все коды нот.

Однако программа должна выбрать одну из мелодий, то есть один из массивов, а затем именно из него извлекать ноты. При обычном способе доступа мы вынуждены конкретно указывать имя массива, с которым мы работаем. И мы не можем для разных мелодий указывать разные массивы. В других языках программирования в таких случаях используют двухмерный массив.

Определение. *Двухмерный массив — это массив, состоящий из нескольких наборов элементов (строк).*

Например, массив `mass[5,10]` будет состоять из пяти строк по десять элементов в каждой. Язык СИ тоже поддерживает двухмерные массивы. Для нашей задачи можно создать массив, имеющий семь строк, в каждой из которых будет храниться одна мелодия. Но в данном случае подобное решение будет иметь один **недостаток**. В двухмерном массиве все строки всегда имеют равную длину. А наши мелодии по длине разные. Конечно, можно выбрать длину для всех строк массива равную длине самой длинной мелодии. Но это приведет к нерациональному использованию памяти.

Применение **ссылочной переменной** сразу решает все проблемы. Каждую мелодию мы помещаем в отдельный одномерный массив типа `unsigned char`. Затем мы создаем ссылочную переменную с именем `nota` (см. **строку 45** нашей программы).

Ну а дальше все просто. Для воспроизведения первой мелодии помещаем в переменную `nota` указатель на начало первого массива. Затем при помощи выражения `*nota` читаем по порядку все элементы массива и воспроизводим ноты, соответствующие прочитанным значениям. После чтения каждого очередного элемента увеличиваем содержимое переменной `nota` на единицу. Если нужно воспроизвести **вторую мелодию**, то записываем в переменную `nota` указатель на начало второго массива. Воспроизводим эту мелодию таким же способом, как и первую. Подобным же образом можно воспроизводить мелодии из любого массива.

Теперь рассмотрим подробнее программу с самого начала.

Листинг 1.18

```
/**************************************************
Project : Prog 9
Version : 1
Date    : 31.01 2006
Author  : Belov
Comments: Музыкальная шкатулка

Chip type           : ATtiny2313
Clock frequency     : 4,000000 MHz
**************************************************/

#include <tiny2313.h>
#include <delay.h>

// Объявление и инициализация массивов

// Таблица задержек
flash unsigned int tabz[] =  {16,32,64,128,256,512,1024};

// Массив коэффициентов деления
flash unsigned int tabkd[] = {0,4748,4480,4228,3992,3768,3556,3356,3168,2990,2822,
       2664,2514,2374,2240,2114,1996,1884,1778,1678,1584,1495,1411,1332,1257,
       1187,1120,1057, 998,942,889,839,792};

// Таблицы мелодий
// В траве сидел кузнечик
flash unsigned char mel1[] = {109,104,109,104,109,108,108,96,108,104,108,104,108,
       109,109,96,109,104,109,104,109,108,108,96,108,104,108,104,108,141,96,109,
       111, 79, 79,111,111,112,80,80,112,112,112,111,109,108,109,109, 96,109,111,
       79,79,111,111,112,80,80,112,112,112,111,109,108,141,128,96,255};
// Песенка крокодила Гены
flash unsigned char mel2[] = {109,110,141,102,104,105,102,109,110,141,104,105,107,
       104,109,110,141,104,105,139,109,110,173,96,114,115,146,109,110,112,109,114,
       115,146,107,109,110,114,112,110,146,109,105,136,107,105,134,128,128,102,105,
       137,136,128,104,107,139,137,128,105,109,141,139,128,110,109,176,112,108,109,
       112,144,142,128,107,110,142,141,128,105,109,139,128,173,134,128,128,109,112,
       144,142,128,107,110,142,141,128,105,109,139,128,173,146,128,255};
// В лесу родилась елочка
flash unsigned char mel3[] = {132,141,141,139,141,137,132,132,132,141,141,142,139,
       176,128,144,146,146,154,154,153,151,149,144,153,153,151,153,181,128,96,255};
// Happy births day to you
flash unsigned char mel4[] = {107,107,141,139,144,143,128,107,107,141,139,146,144,
       128,107,107,151,148,146,112,111,149,117,117,148,144,146,144,128,255};
// С чего начинается родина
flash unsigned char mel5[] = {99,175,109,107,106,102,99,144,111,175,96,99,107,107,
       107,107,102,104,170,96,99,109,109,109,109,107,106,143,109,141,99,109,109,109,
       109,104,106,171,96,99,111,109,107,106,102,99,144,111,143,104,114,114,114,114,
       109,111,176, 96,104,116,112,109,107,106,64,73,143,107,131,99,144,80,80,112,
       111,64,75,173,128,255};
// Из кинофильма "Веселые ребята"
flash unsigned char mel6[] = {105,109,112,149,116,64,80,148,114,64,78,146,112,96,105,
       105,109,144,111,64,80,145,112,64,81,178,96,117,117,117,149,116,64,82,146,112,
       64,79,146,144,96,105,105,107,141,108,109,112,110,102,104,137,128,96,105,105,
       105,137,102,64,73,142,105,107,109,64,75,137,96,105,105,105,137,102,105,142,112,
       64,82,180,96,116,116,116,148,114,112,142,109,64,78,146,144,96,105,105,107,141,
       108,109,112,110,102,104,169,96,96,255};
// Улыбка
flash unsigned char mel7[] = {107,104,141,139,102,105,104,102,164,128,104,107,109,109,
       109,111,114,112,111,109,144,139,128,109,111,144, 96,111,109,104,107,105,173,128,
       111,109,112,107,111,109,109,107,102,104,134,132,128,100,103,107,107,107,107,139,
       112,100,103,102,102,102,134,102,103,107,105,107,108,108,108,108,107,105,107,108,
       144,142,128,112,107,110,140,112,105,108,107,107,107,105,140,139,139,112,103,102,
       103,105,108,107,105,103,128,112,107,110,108,108,108,140,112,105,108,107,107,107,
       139,112,103,102,103,105,108,107,105,103,105,139,132,128,96,96,96,255};

// Таблица начал всех мелодий
flash unsigned char *tabm[] = {mel1, mel2, mel3, mel4, mel5, mel6, mel7};


void main(void)
{
```

```
41  unsigned char count;     // Определяем переменную count
42  unsigned char temp;      // Определяем переменную temp
43  unsigned char fnota;     // Код тона ноты
44  unsigned char dnota;     // Код длительности ноты
45  flash unsigned char *nota;  // Ссылка на текущую ноту

46  PORTB=0x08;    // Инициализация порта PB
47  DDRB=0x08;

48  PORTD=0x7F;    // Инициализация порта PD
49  DDRD=0x00;

50  ACSR=0x80;   // Инициализация (отключение) компаратора

51  TCCR1A=0x00;    // Инициализация таймера счетчика T1
52  TCCR1B=0x09;

53  while (1)
         {
54  m1:    temp=PIND;
55         for (count=0; count<7; count++)  // Цикл сканирования датчиков
              {
56               if ((temp&1)==0) goto m3;    // Проверка младшего бита переменной temp
57               temp >>= 1;                 // Сдвиг содержимого temp
              }
58  m2:    TCCR1A=0x00;           // Выключение звука
59         goto m1;               // Переход на начало

           // Воспроизведение мелодии
60  m3:    nota = tabm[count];                      // Устанавливаем указатель на первую ноту
61  m4:    if (PIND==0x7F) goto m2;                 // Если ни одна кнопка не нажата, закончить
62         if (*nota==0xFF) goto m3;                // Проверка на конец мелодии
63         fnota = (*nota)&0x1F;                    // Определяем код тона
64         dnota = ((*nota)>>5)&0x07;               // Определяем код длительности
65         if (fnota==0) goto m5;                   // Если пауза не воспроизводим звук
66         OCR1A=tabkd[fnota];                      // Программируем частоту звука
67         TCCR1A=0x40;                             // Включаем звук
68  m5:    delay_ms (tabz[dnota]);                  // Формируем задержку
69         TCCR1A=0x00;                             // Выключаем звук
70         delay_ms (tabz[0]);                      // Задержка между нотами
71         nota++;                                  // Перемещаем указатель на следующую ноту
72         goto m4;                                 // К началу цикла
         };
    }
```

Описание программы (листинг 1.18)

Для формирования задержки мы будем использовать функцию из библиотеки delay.h. Поэтому в **строках 1, 2** программы, кроме файла описаний, мы присоединяем и эту библиотеку. Затем начинаются описания всех массивов. В **строке 3** описывается массив, содержащий величины всех музыкальных длительностей.

Так как для формирования длительности мы будем использовать функцию `delay_ms`, величина длительностей задана в миллисекундах. Как видно из текста программы, в данном случае мы используем массив типа `unsigned int`. Переменные этого типа имеют длину два байта, все 16 битов которых используются для хранения информации.

Именно такой тип наиболее подходит для хранения наших коэффициентов. Управляющее слово `fleash` перед описанием массива гарантирует, что эти данные будут размещены в программной памяти микроконтроллера.

В **строках 4, 5, 6** описывается **массив коэффициентов деления для всех нот**. В этом месте программы мы впервые используем перенос строки. Перенос строки применяется в том случае, когда текст команды не помещается в одной строке. Язык СИ разрешает свободно переносить текст на следующую строку. При этом не требуется никаких специальных директив и указателей.

Перенос допускается в том месте команды, где между двумя соседними элементами выражения можно поставить пробел. Тип массива, как и в предыдущем случае, — `usingnerd int`. Содержимое массива `tabkd` полностью соответствует содержимому таблицы с тем же названием из ассемблерного варианта программы.

В **строках 7—38** описываются **семь массивов для хранения семи мелодий**. Массивы имеют тип `unsigned char`. Переменные этого типа занимают в памяти один байт, и все восемь битов этого байта используются для хранения информации. Содержимое каждого из этих массивов полностью соответствует содержимому соответствующих таблиц в ассемблерной версии программы.

В **строке 39** описывается **массив, содержащий адрес начала каждой из семи мелодий**. Это не просто массив, а массив ссылок, на что указывает символ звездочки в тексте его описания. Так же, как и ссылочная переменная, каждый элемент массива ссылок предназначен для хранения ссылки. Данный массив тоже хранится в памяти программ, на что указывает управляющее слово `flesh` в его описании. Элементы этого массива хранят указатели на начало каждого из массивов мелодий, что указано при его инициализации (в фигурных скобках).

Строки 40—72 занимает функция `main`. Начинается функция с описания переменных (**строки 41—45**). Две рабочих переменных `count` и `temp`, а также переменная для хранения кода тона (`tnota`) и переменная для хранения кода длительности (`dnota`) нам уже знакомы. Мы использовали их в предыдущей программе.

Интерес представляет описание переменной `nota`. Это ссылочная переменная, которая предназначена для хранения указателей на объекты в программной памяти, имеющие тип `unsigned char`. Она будет использоваться нами для обращения к элементам массивов, хранящим коды нот. Эти массивы, как уже говорилось, располо-

жены в программной памяти. Поэтому в описании переменной имеется слово `flash`, а перед именем переменной в ее описании стоит символ звездочки. То есть это ссылка на массивы типа `unsigned char`, расположенные во `flesh`.

В **строках 46—52** расположен **блок инициализации**. Эта часть программы полностью повторяет аналогичную часть программы из предыдущего примера (см. **листинг 1.16**).

Строки 53—72 занимает **основной цикл программы**. Цикл состоит всего из двух процедур. В начале цикла (**строки 54—59**) расположена **процедура сканирования клавиатуры**. Эта процедура один к одному скопирована из предыдущего примера (см. **листинг 1.16 строки 14—21**).

При обнаружении нажатой кнопки управление передается по метке `m3` (в новой программе это **строка 60**). Как вы помните, номер нажатой кнопки при выходе из процедуры сканирования содержится в переменной `count`.

Строки 60—72 занимает **процедура проигрывания мелодии**. Проигрывание начинается с того, что в переменную `nota` помещается указатель на массив, содержащий нужную нам мелодию (**строка 60**). А указатель — это элемент массива `tabm`, с номером, равным коду нажатой кнопки. В **строках 61—72** находится цикл, который последовательно считывает мелодию нота за нотой и проигрывает прочитанные ноты. Цикл организован при помощи оператора безусловного перехода (**строка 72**).

Для перемещения вдоль массива содержимое переменной `nota` каждый раз увеличивается на единицу (**строка 71**). В этом же цикле производится проверка состояния клавиатуры (нажата ли еще хоть одна кнопка) и проверка признака конца мелодии. Рассмотрим подробнее, как все это делается.

Проверка состояния клавиатуры происходит в **строке 61**. Если содержимое регистра PIND равно 0x7F, то воспроизведение мелодии прекращается. Управление передается по метке `m2`. Там происходит выключение звука, а затем переход по метке `m1`, то есть к началу основного цикла программы.

Если хоть одна кнопка еще нажата, перехода не происходит и воспроизведение мелодии продолжается. В **строке 62** производится проверка на конец мелодии. Содержимое элемента массива, на который указывает ссылочная переменная `nota` (код ноты), проверяется на равенство числу 0xFF. Если код ноты равен 0xFF, то

управление передается по метке m3, где указатель снова устанавливается на начало мелодии.

В **строке 63** вычисляется значение **кода тона**. Для этого на код ноты, на который указывает переменная nota, накладывается маска. Наложение маски производится при помощи оператора «&». Полученный код тона записывается в переменную fnota.

В **строке 64** производится вычисление **кода длительности**. Для этого применяется составное математическое выражение. Операция (*nota)>>5 сдвигает биты кода ноты на пять шагов вправо. При этом три старших разряда кода становятся тремя младшими. Мы применяем сдвиг вправо потому, что циклический сдвиг влево, использованный нами в Ассемблере, язык СИ не поддерживает. Язык СИ может выполнять только логический сдвиг, но не циклический. На полученное в результате сдвига число налагается маска 0x07. Полученный таким образом код длительности записывается в переменную dnota.

В **строке 65** происходит **проверка кода тона на равенство нулю**. Если код окажется равным нулю, то управление передается по метке m5, то есть к строке, где формируется пауза, обходя строки, где формируется звук.

Звук формируется в **строках 66, 67**. Сначала в регистр совпадения OCR1A помещается коэффициент деления из массива tabkd. Причем указатель массива равен коду тона. Затем в регистр управления TCCR1A записывается код, который подключает таймер к выводу OC1A и, тем самым, включает звук.

В **строке 68** происходит **вызов функции задержки**. В качестве параметра в эту функцию передается коэффициент, извлекаемый из массива tabz. Указатель массива при этом равен коду длительности. После выхода из функции задержки звук выключается.

Для этого в регистр TCCR1A записывается нулевое значение (**строка 69**). В **строке 70** формируется пауза между нотами. В качестве параметра для функции delay_ms в этом случае используется нулевой элемент массива tabz, то есть вырабатывается пауза минимальной длительности.

В **строке 71**, как уже говорилось, происходит **приращение содержимого указателя nota**. Оператор безусловного перехода в **строке 72** замыкает цикл воспроизведения мелодии.

1.11. Кодовый замок

Постановка задачи

Для завершения практикума я подбирал задачу достаточно сложную и интересную, способную как увлечь, так и научить работать с еще неохваченными элементами микроконтроллера. Самым удобным примером, на мой взгляд, является **кодовый замок**. Вообще, микроконтроллеры AVR с их встроенной энергонезависимой памятью (EEPROM) дают широкий простор для разработчика подобных конструкций. Память EEPROM идеально подходит для хранения кода. Причем такой код всегда легко поменять.

При разработке замка мне не хотелось быть тривиальным. Поэтому я предлагаю не совсем обычный замок. Представьте себе кодовый замок, который может воспринимать в качестве кодовой комбинации не только отдельно нажимаемые кнопки, но и любые их сочетания. Например, попарно нажимаемые кнопки, комбинации типа

«Нажать кнопку 6 и, не отпуская, набрать код 257».

И вообще, выбрать любую комбинацию любых кнопок в любом сочетании. Мною был разработан такой замок. Его я и хочу предложить вашему вниманию.

Принцип действия замка следующий: *в режиме записи кода владелец нажимает кнопки набора кода в любом порядке и в любых комбинациях*. Микроконтроллер отслеживает все изменения на клавиатуре и записывает их в ОЗУ. Длина кодовой последовательности ограничена только размерами ОЗУ. Сигналом к окончанию ввода кода служит прекращение манипуляций с клавиатурой.

Считается, что манипуляции закончились, если состояние клавиатуры не изменилось в течение контрольного промежутка времени. Я выбрал его примерно равным одной секунде. Сразу по окончании процесса ввода кода (по окончании контрольного промежутка времени) микропроцессор записывает принятый таким образом код в EEPROM. Код представляет собой последовательность байтов, отражающих все состояния клавиатуры во время набора. После того, как коды будут записаны, замок можно перевести в рабочий режим. Для этого предусмотрен специальный тумблер выбора режимов.

В рабочем режиме замок ждет ввода кода. Для открывания двери необходимо повторить те же самые манипуляции с кнопками, которые вы делали в режиме записи. Микроконтроллер так же, как и в предыдущем случае, отслеживает эти манипуляции и записывает

полученный таким образом код в ОЗУ. По окончании ввода кода (по истечении контрольного промежутка времени) программа переходит в режим сверки кода, находящегося в ОЗУ, и кода, записанного в EEPROM. Сначала сравнивается длина обоих кодов. Затем коды сверяются побайтно. Если сравнение прошло успешно, микроконтроллер подает сигнал на механизм открывания замка.

Итак, **сформулируем задачу** следующим образом:

«Создать схему и программу электронного кодового замка, имеющего десять кнопок для ввода кода, обозначенных цифрами от «0» до «9». Замок должен иметь переключатель режимов «Запись/Работа». В случае правильного набора кода замок должен включать исполнительный механизм замка (соленоид или электромагнитную защелку). Ввод кода должен производиться описанным выше способом».

Алгоритм

При составлении данного алгоритма нам не обойтись без такого понятия, как **«код состояния клавиатуры»**. Что такое код состояния?. Все кнопки клавиатуры подключаются к микроконтроллеру при помощи портов ввода-вывода. Для подключения десяти кнопок (кнопки «0»—«9») одного порта недостаточно. Несколько кнопок придется подключить ко второму.

Контроллер читает содержимое этих портов и получает код, соответствующий их состоянию. Каждой кнопке клавиатуры в этом коде будет соответствовать свой отдельный бит. Когда кнопка нажата, соответствующий бит будет равен нулю. Когда отпущена — единице. Поэтому при разных сочетаниях нажатых и отпущенных кнопок код состояния клавиатуры будет иметь разные значения.

В момент включения питания все кнопки замка должны быть отпущены. Если это не так, то возникает неопределенность в работе замка. Поэтому наш **алгоритм должен начинаться** с процедуры ожидания отпускания всех кнопок. Как только все кнопки окажутся отпущенными или в случае, если они вообще не были нажаты, начинается другая **процедура ожидания**. На сей раз программа ожидает момента нажатия кнопок. Это как раз тот режим работы, в котором замок будет находиться большую часть времени. В момент нажатия любой из кнопок начинается цикл ввода ключевой комбинации.

Процедура ввода ключевой комбинации представляет собой многократно повторяющийся процесс, периодически считывающий код состояния клавиатуры. Каждый раз после очередного считывания

кода программа проверяет, не изменился ли этот код. Как только код изменится, новое его значение записывается в очередную ячейку ОЗУ. В результате, пока состояние клавиатуры не изменяется, программа находится в режиме ожидания.

Как только состояние изменилось, происходит запись нового значения кода состояния в память. Поэтому ключевая комбинация, записанная в ОЗУ, будет представлять собой перечисление всех значений кода состояния клавиатуры, которое он принимал в процессе ввода ключевой комбинации. Длительность же удержания кнопок в каждом из состояний в память не записывается.

Однако это лишь приблизительный алгоритм процедуры ввода ключевой комбинации. Так сказать, ее костяк. На самом деле алгоритм немного сложнее. Обнаружив изменение состояния клавиатуры, программа не сразу записывает новый код в ОЗУ. В целях борьбы с дребезгом контактов, а также для компенсации неточности одновременного нажатия нескольких кнопок, программа сначала выдерживает специальную защитную паузу, затем повторно считывает код состояния клавиатуры и лишь после этого записывает новый код в ОЗУ.

Длительность защитной паузы выбрана равной 48 мс. Такая пауза особенно полезна в случае, если при наборе ключевой комбинации вы хотите использовать одновременное нажатие кнопок. Как бы вы не старались нажать кнопки одновременно, вам этого не удастся. Все равно будет какое-то расхождение в моменте замыкания контактов. Причем порядок замыкания контактов будет зависеть от многих факторов и практически является случайным.

Если не принять специальных мер, то в момент такого нажатия программа зафиксирует не одно, а несколько последовательных изменений кода состояния клавиатуры. Если полученная таким образом кодовая комбинация будет записана в EEPROM, то открыть такой замок будет практически невозможно.

При попытке повторить те же нажатия, замыкание контактов будут происходить в другом порядке. Программа воспримет его как совершенно другой код. Защитная пауза решает эту проблему. В каком бы порядке ни замыкались контакты при одновременном нажатии нескольких кнопок, после паузы все эти процессы закончатся. Повторное считывание даст уже устоявшийся код состояния клавиатуры. Повторить такую комбинацию не составит труда.

Кроме защитной паузы, для борьбы с дребезгом применяется **многократное считывание кода состояния**. То есть на самом деле каждый

раз происходит не одно, а несколько последовательных операций по считыванию кода состояния. Считывание происходит до тех пор, пока несколько раз подряд будет получен один и тот же код.

Теперь поговорим о том, **как программа выходит из процедуры ввода ключевой комбинации**. Как уже говорилось ранее, для выхода из процедуры используется защитный промежуток времени. Для формирования этого промежутка применяется **таймер**. Таймер должен работать в режиме Normal. В этом режиме он просто считает тактовые импульсы.

Процедура ввода кодовой комбинации устроена таким образом, что при каждом нажатии или отпускании любой из кнопок таймер сбрасывается в ноль. В промежутке между нажатиями его показания увеличиваются под действием тактового сигнала. Если в течение защитного промежутка времени не будет нажата ни одна кнопка, показания таймера увеличатся до контрольного предела. Программа постоянно проверяет это условие. Как только показания счетчика превысят контрольный предел, процедура ввода кодовой комбинации завершается. Величина контрольного промежутка времени равна 1 с.

Дальнейшие действия после выхода из процедуры ввода кодовой комбинации определяются состоянием переключателя режимов работы. Если контакты переключателя замкнуты, программа переходит к процедуре записи кодовой комбинации в EEPROM.

Сначала в EEPROM записывается длина кодовой комбинации. А затем байт за байтом и сама комбинация. Если контакты переключателя режима работы разомкнуты, то программа переходит к процедуре проверки кода. Эта процедура сначала извлекает из EEPROM записанную ранее длину кодовой комбинации и сравнивает ее с длиной только что введенной комбинации.

Если две эти величины не равны, процедура проверки кода сразу же завершается с отрицательным результатом. Если длина обеих комбинаций одинакова, то программа приступает к побайтному их сравнению. Для этого она поочередно извлекает из EEPROM ранее записанные туда байты и сравнивает каждый из них с соответствующими байтами в ОЗУ. При первом же несовпадении процесс сравнения также завершается. И завершается отрицательно.

И только в том случае, если все байты в ОЗУ и в EEPROM окажутся одинаковыми, сравнение считается успешным. В случае успешного сравнения программа переходит к **процедуре открывания замка**. Процедура открывания начинается с выдачи открывающего сигнала

на исполнительный механизм. Затем программа выдерживает паузу в 2 с и снимает сигнал. Этого времени достаточно для того, чтобы открыть дверь. Затем замок переходит в исходное состояние.

Схема

Возможный вариант схемы замка приведен на **рис. 1.17**. Кнопки S1—S10 служат для набора кода. Переключатель S11 предназначен для выбора режима работы. Если контакт переключателя S11 замкнут, замок переходит в режим «Запись». Разомкнутые контакты соответствуют режиму «Работа».

Схема управления механизмом замка состоит из транзисторного ключа VT1 и электромагнитного реле K1. Резистор R1 ограничивает ток базы ключа. Диод VD1 служит для защиты от напряжения самоиндукции, возникающей на катушке реле. Питание реле осуществляется от отдельного источника +12 В (питание микроконтроллера +5 В). Если в качестве VT1 применять транзистор КТ315, то электромагнитное реле может иметь рабочее напряжение +12 В и рабочий ток не более 250 мА. Контакты реле должны быть рассчитаны на управление исполнительным механизмом (соленоидом).

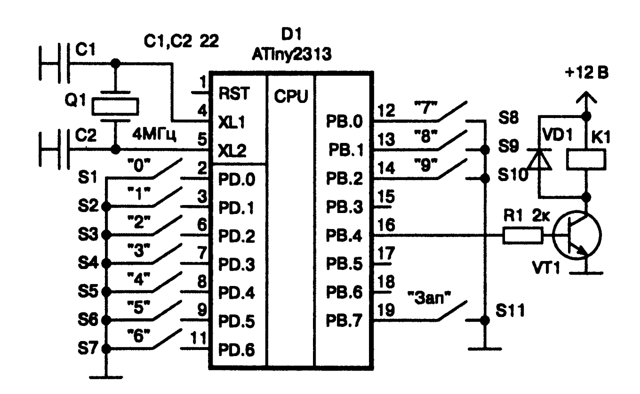


Рис. 1.17. *Схема кодового замка*

Обратите внимание, что в данной схеме одни линии порта PB будут работать как входы, а другие (в частности линия PB.4) — как выходы. При распределении выводов порта между периферийными

устройствами учитывалась возможность объединения замка с музыкальной шкатулкой. В этом случае шкатулка может управляться одной кнопкой и служить дверным звонком.

Программа на Ассемблере

Один из возможных вариантов программы на Ассемблере приведен **в листинге 1.19**. Прежде чем переходить к подробному описанию ее работы, разберемся в некоторых общих вопросах. Начнем с **кода состояния клавиатуры**. Как уже говорилось ранее, код состояния должен иметь отдельный бит для каждой кнопки клавиатуры. Итого, получается десять битов.

Одного байта явно мало. Значит, мы должны использовать двухбайтовый код состояния. Самый простой способ получить двухбайтовый код состояния — это прочитать сначала содержимое порта PD, а затем — содержимое порта PB. Затем нужно наложить на каждый из полученных байтов маску.

Маска должна обнулить все ненужные нам разряды и оставить разряды, к которым подключены наши кнопки. Для числа, прочитанного из порта PD, маска должна обнулить самый старший разряд и оставить все остальные. Для числа, прочитанного из порта PB, нужно, напротив, оставить три младших разряда и обнулить все остальные. Полученные таким образом два байта мы и будем считать кодом состояния клавиатуры.

В том случае, если все десять кнопок (S1—S10) отпущены, код состояния клавиатуры равен 0x7F, 0x07 (0b01111111, 0b00000111). В таком коде значащие биты, отражающие состояние той или иной кнопки, равны единице, а все остальные биты равны нулю. Если нажать любую кнопку, то код состояния изменится. Соответствующий этой кнопке бит примет нулевое значение. Таким образом, любое изменение состояния клавиатуры вызовет соответствующее изменение кода.

Теперь **вернемся к тексту программы**. В программе применяются следующие новые для нас операторы.

cli

Общий запрет прерываний. Действие данной команды обратно действию уже знакомой нам команды `sei`. Команда не имеет параметров и служит для сброса флага I в регистре SREG.

st

Косвенная запись в память. Команда имеет три модификации:

```
st W,Rd    st W+,Rd    st -W,Rd,
```

где W — это одна из регистровых пар (X, Y или Z). Rd — имя одного из регистров общего назначения. Независимо от модификации команда выполняет запись содержимого регистра Rd в ОЗУ по адресу, который хранится в регистровой паре W.

При этом **первая модификация команды** не изменяет содержимое регистровой пары W. **Вторая модификация** увеличивает содержимое регистровой пары на единицу после того, как произойдет запись. А **третья модификация** команды уменьшает на единицу содержимое регистровой пары перед тем, как произойдет запись в ОЗУ.

В **строке 75** нашей программы (см. **листинг 1.19**) содержимое регистра `XL` записывается в ОЗУ по адресу, который хранится в регистровой паре Z. После этого содержимое регистровой пары Z увеличивается на единицу.

ld

Косвенное чтение из памяти. Данная операция является обратной по отношению к предыдущей. Она тоже имеет три модификации:

```
ld Rd,W    ld Rd,W+    ld Rd,-W
```

Операция производит чтение байта из ячейки ОЗУ, адрес которой хранится в регистровой паре W (то есть X, Y или Z) и записывает прочитанный байт в регистр общего назначения Rd. Содержимое регистровой пары так же, как и в предыдущем случае, ведет себя поразному, в зависимости от модификации команды. То есть оно либо не изменяется, либо увеличивается после чтения, либо уменьшается прежде, чем байт будет прочитан.

В **строке 97** нашей программы (**листинг 1.19**) читается байт из ячейки ОЗУ, адрес которой хранится в регистровой паре Z, и записывается в регистр `data`. Затем содержимое регистровой пары Z увеличивается на единицу.

brsh

Переход по условию «больше или равно». В качестве условия для перехода выступает содержимое флага переноса С. Флаг переноса устанавливается по результатам операции сравнения или вычитания. Команда имеет всего один параметр — **относительный адрес перехода**. Переход выполняет в том случае, если флаг переноса равен нулю. А это происходит только тогда, когда в предшествующей операции сравнения (вычитания) второй операнд окажется больше или равен первому.

sbic

Оператор типа «проверить — пропустить». Общая форма записи команды: `sbic A,n`,

где `A` — номер регистра ввода—вывода;

`n` — номер разряда.

Вместо номера регистра и номера разряда может использоваться имя регистра и имя разряда. Обычно используются стандартные имена от фирмы Atmel. Команда проверяет содержимое разряда номер `n` регистра A. Если разряд сброшен, то очередная команда программы не выполняется.

Пример использования данной команды — **строка 159** нашей программы (см. **листинг 1.19**). В этой строке команда `sbic` проверяет бит `EEWE` регистра `EECR`. Если этот бит сброшен, то команда в **строке 160** не выполняется, а управление передается к **строке 161**. Если бит установлен, то выполняется команда в **строке** 160. Команда `sbic` имеет одно ограничение. Она работает с регистрами ввода—вывода с адресами в диапазоне от 0 до 31.

cbr

Сброс разрядов РОН. Данный оператор предназначен для одновременного сброса нескольких разрядов. Оператор имеет два параметра. **Первый параметр** — это имя регистра общего назначения, разряды которого должны быть сброшены. **Второй параметр** — это маска сброса разрядов. В данном случае маска — это двоичное число, у которого в единицу установлены те разряды, которые должны быть сброшены. Например, в **строке 129** программы (**листинг 1.19**) сбрасываются разряды регистра XH. Значение маски равно 0xF8. В двоичном виде число 0xF8 выглядит так: 0b11111000. Поэтому в результате действия

команды `cbr` в **строке 129** пять старших разрядов числа, находящегося в регистре XH, будут сброшены в ноль, а три младшие останутся без изменений. Это альтернативный способ наложения маски.

Листинг 1.19

```
     ;############################################
     ;##              Пример 10                 ##
     ;##            Кодовый замок               ##
     ;############################################

     ;------------------------ Псевдокоманды управления

1    .include "tn2313def.inc"   ; Присоединение файла описаний
2     list                      ; Включение листинга

     ;------------------------Модуль описаний

3    .def     drebL = R1        ; Буфер антидребезга младший байт
4    .def     drebH = R2        ; Буфер антидребезга старший байт
5    .def     temp = R16        ; Вспомогательный регистр
6    .def     data = R17        ; Регистр передачи данных
7    .def     flz = R18         ; Флаг задержки
8    .def     count = R19       ; Регистр передачи данных
9    .def     addre = R20       ; Указатель адреса в EEPROM
10   .def     codL = R21        ; Временный буфер кода младший байт
11   .def     codH = R22        ; Временный буфер кода старший байт

     ;------------------------ Определение констант

12   .equ     bsize = 60        ; Размер буфера для хранения кода
13   .equ     kzad  = 3000      ; Константа, определяющая длительность защитной паузы
14   .equ     kandr = 20        ; Константа антидребезга

     ;------------------------ Резервирование ячеек памяти (SRAM)

15            .dseg             ; Выбираем сегмент ОЗУ
16            .org     0x60     ; Устанавливаем текущий адрес сегмента

17   bufr:    .byte    bsize    ; Буфер для приема кода

     ;------------------------ Резервирование ячеек памяти  (EEPROM)

18            .eseg             ; Выбираем сегмент EEPROM
19            .org     0x08     ; Устанавливаем текущий адрес сегмента

20   klen:    .byte    1        ; Ячейка для хранения длины кода
21   bufe:    .byte    bsize    ; Буфер для хранения кода

     ;------------------------ Начало программного кода

22            .cseg             ; Выбор сегмента программного кода
23            .org     0        ; Установка текущего адреса на ноль

24   start:   rjmp     init     ; Переход на начало программы
25            reti              ; Внешнее прерывание 0
26            reti              ; Внешнее прерывание 1
27            reti              ; Таймер/счетчик 1, захват
28            rjmp     propr    ; Таймер/счетчик 1, совпадение, канал А
29            rjmp     propr    ; Таймер/счетчик 1, прерывание по переполнению
30            reti              ; Таймер/счетчик 0, прерывание по переполнению
31            reti              ; Прерывание UART прием завершен
32            reti              ; Прерывание UART регистр данных пуст
33            reti              ; Прерывание UART передача завершена
34            reti              ; Прерывание по компаратору
35            reti              ; Прерывание по изменению на любом контакте
36            reti              ; Таймер/счетчик 1. Совпадение, канал В
37            reti              ; Таймер/счетчик 0. Совпадение, канал В
38            reti              ; Таймер/счетчик 0. Совпадение, канал А
39            reti              ; USI готовность к старту
```

```
        reti                  ; USI Переполнение
        reti                  ; EEPROM Готовность
        reti                  ; Переполнение охранного таймера

;***************************************************
;*                                                 *
;*              Модуль инициализации               *
;*                                                 *
;***************************************************

init:
;------------------------- Инициализация стека

        ldi     temp, RAMEND    ; Выбор адреса вершины стека
        out     SPL, temp       ; Запись его в регистр стека

;------------------------- Инициализация портов В/В

        ldi     temp,0x18       ; Инициализация порта PB
        out     DDRB, temp
        ldi     temp,0xE7
        out     PORTB, temp

        ldi     temp, 0x7F      ; Инициализация порта PD
        out     PORTD, temp
        ldi     temp, 0
        out     DDRD, temp

;-------------------------- Инициализация (выключение) компаратора

        ldi     temp, 0x80
        out     ACSR, temp

;-------------------------- Инициализация таймера

        ldi     temp, high(kzad )      ; Записываем коэффициент задержки
        out     OCR1AH, temp
        ldi     temp, low(kzad )
        out     OCR1AL, temp
        ldi     temp, 0x03             ; Выбор режима работы таймера
        out     TCCR1B, temp

;***************************************************
;*                                                 *
;*           Начало основной программы             *
;*                                                 *
;***************************************************

main:   ldi     codL,0x7F       ; Код для сравнения (младший байт)
        ldi     codH,0x07       ; Код для сравнения (старший байт)

m0:     rcall   incod           ; Ввод и проверка кода клавиш
        brne    m0              ; Если хоть одна не нажата, продолжаем

m1:     rcall   incod           ; Ввод и проверка кода клавиш
        breq    m1              ; Если не одна не нажата, продолжаем

m2:     ldi     ZH, high(bufr)   ; Установка указателя на начало буфера
        ldi     ZL, low(bufr)    ;
        clr     count            ; Сброс счетчика байт

;--------------------- Цикл ввода кода

m3:     cli                     ; Запрет всех прерываний
        ldi     data,1          ; Вызываем задержку первого типа
        rcall   wait            ; К подпрограмме задержки

m5:     rcall   incod           ; Ввод и проверка кода кнопок
        st      Z+,XL           ; Записываем его в буфер
        st      Z+,XH
        inc     count           ; Увеличение счетчика байтов
        inc     count
        cpi     count, bsize    ; Проверяем, не конец ли буфера
        brsh    m7              ; Если конец, завершаем ввод кода
```

```
        mov     codL, XL        ; Записываем код как старый
        mov     codH, XH

        ldi     data,2          ; Вызываем задержку третьего типа
        rcall   wait            ; К подпрограмме задержки

m6:     rcall   incod           ; Ввод и проверка кода кнопок
        brne    m3              ; Если изменилось, записываем в буфер
        cpi     flz,1           ; Проверка окончания фазы ввода кода
        brne    m6

; ---------------------------- Опрос состояния тумблера

m7:     sbic    PINB,7          ; Проверка состояния тумблера
        rjmp    m9              ; К процедуре проверки кода

; ---------------------------- Процедура записи кода

        mov     data, count     ; Помещаем длину кода в data
        ldi     addre, klen     ; Адрес в EEPROM для хранения длины кода
        rcall   eewr            ; Записываем в длину кода EEPROM

        ldi     addre, bufe       ; В регистр адреса начало буфера в EEPROM
        ldi     ZH, high(bufr)    ; В регистровую пару Z записываем
        ldi     ZL, low(bufr)     ; адрес начала буфера в ОЗУ

m8:     ld      data, Z+        ; Читаем очередной байт из ОЗУ
        rcall   eewr            ; Записываем в длину кода EEPROM
        dec     count           ; Декремент счетчика байтов
        brne    m8              ; Если не конец, продолжаем запись

        rjmp    m11             ; К процедуре открывания замка

; ---------------------------- Процедура проверки кода

m9:     ldi     addre, klen     ; Адрес хранения длины кода
        rcall   eerd            ; Чтение длины кода из EEPROM
        cp      count, data     ; Сравнение с новым значением
        brne    m13             ; Если не равны, к началу

        ldi     addre, bufe       ; В YL начало буфера в EEPROM
        ldi     ZH, high(bufr)    ; В регистровую пару Z записываем
        ldi     ZL, low(bufr)     ; адрес начала буфера в ОЗУ

m10:    rcall   eerd            ; Читаем очередной байт из EEPROM
        ld      temp, Z+        ; Читаем очередной байт из ОЗУ

        cp      data, temp      ; Сравниваем два байта разных кодов
        brne    m13             ; Если не равны, переходим к началу

        dec     count           ; Уменьшаем содержимое счетчика байтов
        brne    m10             ; Если не конец, продолжаем проверку

; ---------------------------- Процедура открывания замка

m11:    sbi     PORTB,4         ; Команда "Открыть замок"
        ldi     data,3          ; Вызываем задержку третьего типа
        rcall   wait
        cbi     PORTB,4         ; Команда "Закрыть замок"

m13:    rjmp    main            ; Перейти к началу

;*****************************************************
;*                                                   *
;*            Вспомогательные процедуры              *
;*                                                   *
;*****************************************************

; ----------------------------- Ввод и проверка 2 байтов с клавиатуры

incod:  push    count

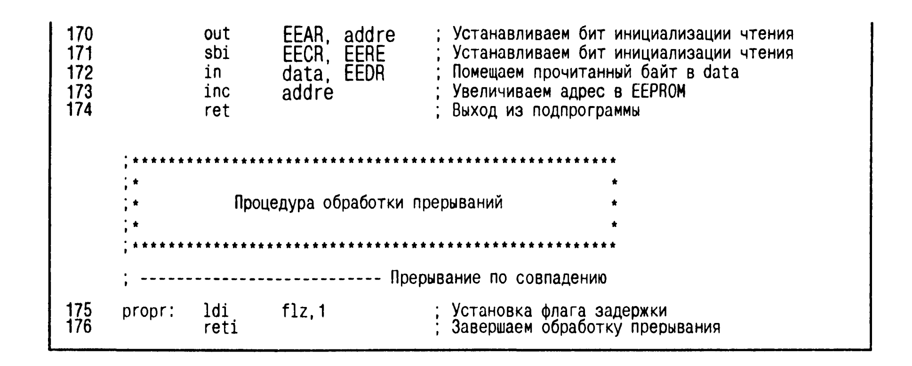
        ldi     XL,0            ; Обнуление регистровой пары X
```

```
            ldi     XH,0
ic1:        ldi     count, kandr    ; Константа антидребезга
            mov     drebL, XL       ; Старое значение младший байт
            mov     drebH, XH       ; Старое значение старший байт
ic2:        in      XL,PIND         ; Вводим код (младший байт)
            cbr     XL,0x80         ; Накладываем маску
            in      XH,PINB         ; Вводим код (старший байт)
            cbr     XH,0xF8         ; Накладываем маску
ic3:        cp      XL, drebL       ; Сверяем младший байт
            brne    ic1             ; Если не равен, начинаем с начала
            cp      XH, drebH       ; Сверяем старший байт
            brne    ic1             ; Если не равен, начинаем с начала
ic4:        dec     count           ; Уменьшаем счетчик антидребезга
            brne    ic2             ; Если еще не конец, продолжаем
            cp      XL, codL        ; Сравнение с временным буфером
            brne    ic5             ; Если не равно, заканчиваем сравнение
            cp      XL, codH        ; Сравниваем старшие байты
ic5:        pop     count
            ret

; ------------------------------ Подпрограмма задержки
wait:       cpi     data,1          ; Проверяем код задержки
            brne    w1
            ldi     temp,0x40       ; Разрешаем прерывание по совпадению
            rjmp    w2
w1:         ldi     temp,0x80       ; Разрешаем прерывания по переполнению
w2:         out     TIMSK,temp      ; Записываем маску
            clr     temp
            out     TCNT1H,temp     ; Обнуляем таймер
            out     TCNT1L,temp
            ldi     flz,0           ; Сбрасываем флаг задержки
            sei                     ; Разрешаем прерывания
            cpi     data,2          ; Если это задержка 2-го типа
            breq    w4              ; Переходим к концу подпрограммы
w3:         cpi     flz,1           ; Ожидание окончания задержки
            brne    w3
            cli                     ; Запрещаем прерывания
w4:         ret                     ; Завершаем подпрограмму

; ------------------------------ Запись байта в ячейку EEPROM
eewr:       cli                     ; Запрещаем прерывания
            sbic    EECR, EEWE      ; Проверяем готовность EEPROM
            rjmp    eewr            ; Если не готов ждем
            out     EEAR, addre     ; Записываем адрес в регистр адреса
            out     EEDR, data      ; Записываем данные в регистр данных
            sbi     EECR, EEMWE     ; Устанавливаем бит разрешения записи
            sbi     EECR, EEWE      ; Устанавливаем бит записи
            inc     addre           ; Увеличиваем адрес в EEPROM
            ret                     ; Выход из подпрограммы

; ------------------------------ Чтение байта из ячейки EEPROM
eerd:       cli                     ; Запрещаем прерывания
            sbic    EECR, EEWE      ; Проверяем готовность EEPROM
            rjmp    eerd            ; Если не готов ждем
```



```
170          out    EEAR, addre    ; Устанавливаем бит инициализации чтения
171          sbi    EECR, EERE     ; Устанавливаем бит инициализации чтения
172          in     data, EEDR     ; Помещаем прочитанный байт в data
173          inc    addre          ; Увеличиваем адрес в EEPROM
174          ret                   ; Выход из подпрограммы

     ;*****************************************************
     ;*                                                   *
     ;*         Процедура обработки прерываний            *
     ;*                                                   *
     ;*****************************************************

     ; -------------------------- Прерывание по совпадению

175  propr:  ldi    flz,1          ; Установка флага задержки
176          reti                  ; Завершаем обработку прерывания
```

Описание программы (листинг 1.19)

Строки 1, 2 программы, я думаю, вопросов не вызывают. В **строках 3—11** происходит описание всех используемых в программе переменных. Назначение каждой из этих переменных мы рассмотрим в ходе описания принципов работы программы. Далее, в **строках 12—14** происходит описание констант. Каждая из трех констант определяет один из параметров нашего устройства. Остановимся на этом подробнее.

Константа `bsize` (строка 12) определяет **размер буфера для хранения кодовой комбинации**. Этот размер выбран равным 60 ячейкам. Учитывая, что в буфер будут записываться коды состояния клавиатуры, а каждый такой код состоит из двух байтов, в буфер указанного размера можно записать последовательность из 30 кодов.

Если учитывать, что записи подлежит каждое изменение кода состояния, а его изменение происходит как при нажатии кнопки, так и при ее отпускании, то после нажатия и отпускания одной из кнопок в буфер запишется два кода.

Значит, объема буфера нам хватит на 15 последовательных нажатий. Этого вполне достаточно, так как типичная кодовая комбинация состоит обычно из 4—5 цифр. Если вы считаете, что этого недостаточно, вы можете увеличить размер буфера, просто поменяв значение константы `bsize` в **строке 12**. Максимально возможный размер ограничен объемом ОЗУ и равен примерно 100 байтам (полный размер ОЗУ 128 байт).

Учтите, что буфер не может занимать весь объем ОЗУ, так как в верхних адресах необходимо обязательно оставить пространство, которое будет использовать стековая память. Обращаю ваше внимание, что константа `bsize` используется для задания размера не только буфера в ОЗУ, но и для задания размера буфера в EEPROM, который предназначен для долговременного хранения кодовой комбинации.

Константа `kzad` (строка 13) определяет **длительность защитной паузы.** Назначение защитной паузы подробно описано выше. Константа представляет собой коэффициент пересчета для таймера, при котором величина сформированной задержки будет равна 48 мс.

Константа `kandr` (строка 14) — это **константа антидребезга**. Она используется в специальной антидребезговой процедуре. Константа определяет, сколько раз подряд должен повториться один и тот же код состояния клавиатуры, чтобы программа прекратила цикл антидребезга и перешла к обработке считанного кода.

После определения констант начинается **блок резервирования оперативной памяти (строки 15—17)**. В **строке 15** выбирается соответствующий сегмент памяти, в **строке 16** устанавливается указатель на адрес 0x60. Соображения для выбора именно этого адреса уже приводились в предыдущем примере.

Собственно резервирование ячеек производится в **строке 17**. Директива `byte` резервирует необходимое количество ячеек ОЗУ, начиная с адреса, определяемого меткой `bufr`. В данном случае `bufr` будет равен 0x60. Количество резервируемых ячеек определяется константой `bsize`.

Далее, в **строках 18—21** происходит резервирование ячеек в энергонезависимой памяти (EEPROM). В **строке 18** выбирается сегмент EEPROM. **В строке 19** устанавливается текущее значение указателя этого сегмента. Указателю присваивается значение 0x08. То есть размещение данных в памяти EEPROM будет начинаться с восьмой ячейки. Вообще-то в данном случае можно было начинать с нулевой ячейки. Это не сделано лишь из соображений надежности работы.

Фирма *Atmel* не рекомендует без особой необходимости использовать ячейку с нулевым адресом, так как именно она подвергается наибольшему риску потери информации при недопустимых перепадах напряжения питания, особенно если перепады напряжения возникают в момент записи информации в EEPROM. Так как EEPROM не работает со стеком, у нас есть запас по ячейкам. Поэтому мы отступили на целых восемь ячеек.

Собственно команды резервирования занимают **строки 20** и **21**. В **строке 20** резервируется одна ячейка, в которой будет храниться длина ключевой комбинации. В **строке 21** резервируется буфер длиной `bsize`, в котором будет храниться сама комбинация.

После резервирования ячеек мы переходим **в сегмент программного кода** (**строка 22**). И начинаем формирование программы с нулевого адреса (команда `org` в **строке 23**). Программный код начинается с **таблицы переопределения векторов прерываний** (**строки 24—42**). Как видите, в данном случае мы будем использовать два вида прерываний.

Это прерывание по совпадению в канале А таймера/счетчика 1 (**строка 28**) и прерывание по переполнению того же таймера (**строка 29**). Первое прерывание используется для формирования защитной задержки в 48 мс. А второе — для формирования контрольного промежутка времени в 1 с. Для обоих видов прерываний назначена одна и та же процедура обработки: `propr`. Почему одна и та же и как она работает, мы узнаем чуть позже.

В **строках 44—61** расположен **модуль инициализации**. Начинается инициализация с программирования портов ввода—вывода (**строки 46—53**). Все разряды порта PD и большая часть разрядов порта PB конфигурируются как входы. И лишь два разряда PB.3 и PB.4 конфигурируются как выходы.

Разряд PB.4 используется для управления механизмом замка. А **разряд PB.3** вообще пока не используется (зарезервирован). Его мы будем использовать как выход звука, когда будем объединять в одно устройство наш кодовый замок и музыкальную шкатулку. Для всех разрядов обоих портов, настроенных на ввод, включаются внутренние нагрузочные резисторы. На обоих выходах (PB.3 и PB.4) устанавливается низкий логический уровень.

В **строках 54, 55** программируется **аналоговый компаратор**. Программирование таймера/счетчика 1 производится в **строках 56—61**. Сначала в обе половинки регистра совпадения (OCR1AH, OCR1AL) записывается код, определяющий длительность защитной задержки (**строки 56—59**). Затем в регистр состояния TCCR1B записывается код 0x03 (**строки 60, 61**). При записи этого кода таймер/счетчик 1 переводится в режим Normal с использованием предварительного делителя. А коэффициент предварительного деления становится равным 1/64.

Строки 62—119 занимает **основной цикл программы, строки 120—174 — набор вспомогательных подпрограмм**, а **строки 175 и**

176 — процедура обработки прерываний. Рассмотрение программы удобнее начать со вспомогательных процедур.

И первая процедура, которую мы рассмотрим — это **процедура ввода и предварительной обработки кода состояния клавиатуры**. Процедура представляет собой подпрограмму с именем `incod` и расположена в **строках 120—140**. В процессе работы процедура использует регистровую пару X и две вспомогательные регистровые пары: codH, codL (описаны в **строках 10, 11**) и drebH, drebL (описаны в **строках 3, 4**).

Главная задача данной процедуры — получить **код состояния клавиатуры**, используя алгоритм многократного считывания для борьбы с антидребезгом. Кроме того, процедура выполняет еще одну, вспомогательную функцию. Она сравнивает полученный описанным выше способом код состояния клавиатуры с другим кодом, который хранится в паре регистров codH—codL. Такое сравнение используется в основной программе для оценки значения кода состояния.

Таким образом, процедура `incod` по окончании своей работы возвращает два разных значения. Во-первых, код состояния клавиатуры (в регистровой паре X), а во-вторых, результат сравнения кода с контрольным значением (используя флаг нулевого результата Z).

Рассмотрим работу процедуры подробнее. Во-первых, подпрограмма использует традиционное сохранение и восстановление содержимого регистров в стеке. На этот раз сохранению подлежит всего один регистр — регистр `count`. В **строке 120** его значение сохраняется, а в **строке 139** — восстанавливается. Основной же текст подпрограммы состоит из двух частей:

- в первой части (**строки 121—133**) реализуется алгоритм ввода кода и борьбы с антидребезгом;
- во второй части (**строки 134—138**) производится сравнение полученного кода с числом в буфере codH-codL.

Начнем с **процедуры ввода кода и борьбы с антидребезгом**. Эта процедура представляет собой бесконечный цикл, который при каждом проходе производит формирование кода состояния клавиатуры и при этом подсчитывает, сколько раз подряд полученный код будет иметь одинаковое значение. Для подсчета используется регистр `count`. Причем в начале в `count` записывается константа `kandr`, и каждый раз, когда новый код равен предыдущему, содержимое `count` уменьшается на единицу (используется обратный счет).

Если при очередном проходе код состояния изменит свое значение, то в регистр `count` снова записывается константа `kandr`, и подсчет начинается сначала. Заканчивается цикл считывания кодов лишь тогда, когда значение `count` достигнет нуля. Новое значение кода состояния клавиатуры формируется в регистровой паре X. Для того, чтобы новое значение можно было сравнивать со старым, старое записывается в буфер drebH—drebL.

Теперь **текст описанной выше программы рассмотрим по порядку**. Все начинается с подготовки регистровой пары X к принятию нового значения кода. В **строках 121, 122** содержимое X обнуляется. Бесконечный цикл многократного считывания занимает **строки 123—135**.

В **строке 123** в регистр `count` записывается начальное значение `kandr`. В **строках 124, 125** содержимое регистровой пары X сохраняется в буфере drebH—drebL. Таким образом запоминается старое значение кода состояния перед тем, как будет получено новое. В **строках 126—129** происходит считывание кодов из портов PD и PB и наложение масок. Полученный в результате этих операций код состояния клавиатуры окажется в регистровой паре X.

В **строках 130—133** производится сравнения старого и нового значений кодов. Сравнение происходит побайтно. Сначала сравниваются младшие байты (**строки 130, 131**), затем старшие (**строки 132, 133**). Оператор `brne` выполняет переход по условию «не равно». Поэтому, если хотя бы одна из операций сравнения даст положительный результат (коды окажутся неодинаковыми), то управление перейдет по метке `ic1`. То есть к тому месту программы, где счетчику `count` снова присваивается значение `kandr`.

Если же обе операции сравнения дадут отрицательные результаты,и коды окажутся равными, управление перейдет к **строке 134**. В **строке 134** происходит уменьшение содержимого регистра `count` на единицу. После этого производится проверка на ноль (**строка 135**). Если после уменьшения содержимого `count` оно еще не равно нулю, то оператор `brne` в **строке 135** передает управление по метке `ic2`, и цикл антидребезга продолжается. В противном случае цикл завершится, и управление переходит к **строке 136**.

В **строках 136—138** находится процедура сравнения только что полученного значения кода состояния клавиатуры с числом в буфере codH—codL. Сравнение проходит побайтно. Сначала в **строке 136** сравниваются младшие байты. Если они не равны, то дальнейшее сравнение не имеет смысла. Поэтому управление передается по метке `ic5`, и подпрограмма завершается.

Если младшие байты сравниваемых величин оказались равны, то окончательный результат сравнения теперь можно получить, просто сравнив между собой старшие байты. Это сравнение производится в **строке 138**. В результате, при выходе из подпрограммы `incod`:

- флаг Z будет установлен, если сравниваемые коды равны между собой;
- флаг Z будет сброшен, если коды не равны.

Следующая дополнительная процедура, обеспечивающая работу основной части программы, — это **подпрограмма формирования задержки**. Для формирования временных интервалов эта процедура использует таймер. Мы уже рассматривали два варианта подобных процедур. Одна из них использовала прямое чтение содержимого таймера и цикл ожидания, а вторая использовала прерывания.

В данном случае разработан еще один вариант, который представляет собой нечто среднее между двумя предыдущими. В этом варианте будут использоваться и прерывание, и цикл ожидания. Новый алгоритм реализует подпрограмма `wait`, которая занимает **строки 141—157**. Подпрограмма имеет три режима работы. Номер режима передается в подпрограмму при ее вызове.

Для этого он записывается в регистр `data`. В **режиме номер 1** подпрограмма формирует задержку 48 мс. В **режиме номер 3** формируется задержка в 1 с. В **режиме номер 2** задержка не формируется. Подпрограмма просто настраивает таймер точно так же, как в режиме номер 3, разрешает прерывания и заканчивает свою работу. Контрольный интервал времени длительностью 1 с формируется уже вне подпрограммы `wait` (в основном тексте программы).

Основной **принцип формирования задержки** строится на использовании флага задержки. В качестве флага задержки применяется регистр `flz`. Описание этого регистра вы можете видеть в **строке 7** программы. Перед началом цикла задержки в регистр `flz` записывается ноль. Затем запускается таймер и разрешается работа одного из видов прерываний (прерывание по совпадению или прерывание по переполнению).

В определенный момент времени будет вызвана **процедура обработки прерывания**. Эта процедура (**строки 156, 176**) запишет в регистр флага (`flz`) единицу. Единица в регистре `flz` послужит индикатором того факта, что заданный промежуток времени закончился. Для обнаружения момента окончания задержки используется цикл ожидания.

В цикле ожидания программа постоянно проверяет содержимое регистра flz. Пока flz равно нулю, цикл продолжается. Заканчивается цикл в тот момент, когда flz будет равен единице.

Для формирования разных значений длительности задержки используются разные виды прерываний. Прерывание по совпадению используется для формирования задержки в 48 мс. Для этого значение регистра совпадения выбрано таким образом, чтобы содержимое счетного регистра достигло этого значения именно через 48 мс. Прерывание по переполнению таймера используется для формирования интервала времени, равного одной секунде. Благо, что при коэффициенте пересчета предварительного делителя 1/64 и тактовой частоте в 4 МГц переполнение таймера произойдет именно через 1 с.

Рассмотрим **текст подпрограммы wait** подробнее. Начинается подпрограмма с проверки значение регистра data (**строка 141**). Как уже говорилось ранее, при помощи этого регистра в подпрограмму передается код номера режима. В данном случае нужно определить длительность формируемой задержки.

От этого зависит, какой из видов прерываний будет активизирован. Для активизации того либо другого вида прерываний в регистр маски таймера (TIMSK) мы будем записывать разные значения маски. В **строках 141—145** как раз и выбирается это значение. Выбор сводится к проверке номера режима.

Если содержимое data равно 1, то выполняется **строка 143**, где в качестве маски выбирается число 0x40 (прерывание по совпадению). Выбранная маска записывается в регистр temp. Оператор безусловного перехода rjmp в **строке 144** передает управление по метке w2 для того, чтобы «перепрыгнуть» **строку 145**, где выбирается другое значение маски.

Если код в регистре data не равен 1, то управление передается к **строке 145**, где в регистр temp записывается код 0x80. Маска 0x80 разрешает прерывание по переполнению. Подробнее о настройке таймера смотрите в **главе 6**. В **строке 146** выбранное значение маски записывается в регистр TIMSK.

В **строках 147—149** выполняется **сброс таймера** (в обе половины счетного регистра таймера записывается нулевое значение). С этого момента начинается отсчет времени задержки. В **строке 150** сбрасывается в ноль регистр флага задержки (flz). А в **строке 151** производится глобальное разрешение прерываний. На этом все настройки, необходимые для работы процедуры задержки, заканчиваются.

Остается лишь ждать окончания заданного промежутка времени. Но прежде, чем начинать цикл ожидания, программа производит еще одну проверку номера режима (**строки 152, 153**). Дело в том, что для режима номер 2 цикл ожидания организовывать не нужно. В этом случае цикл ожидания располагается вне подпрограммы `wait`, в теле самой программы. Поэтому, если был задан режим номер 2, то работа подпрограммы в этом месте должна заканчиваться.

Оператор `cpi` в **строке** 152 проверяет содержимое регистра `data`, где хранится код номера режима на равенство цифре 2. Если код режима равен двум, то оператор условного перехода в **строке 153** передает управление на конец подпрограммы. В противном случае программа переходит к циклу ожидания.

Цикл ожидания занимает всего две **строки (154, 155)**. В **строке 154** производится проверка содержимого регистра флага (`flz`) на равенство единице. Пока счетчик находится в процессе счета и прерывание еще не сработало, то содержимое регистра `flz` равно нулю, оператор в **строке 155** передает управления назад на **строку 154**, и цикл продолжается. Как только процедура обработки прерывания запишет в `flz` единицу, цикл завершается, и управление переходит к **строке 156**. В этой **строке** происходит глобальный запрет всех прерываний. А в **строке 157** подпрограмма `wait` завершается.

Две оставшиеся, еще не описанные вспомогательные процедуры предназначены для работы с EEPROM. При записи в этот вид памяти и чтения из нее нужно соблюдать определенную последовательность действий. Эта последовательность подробно описана в документации на микроконтроллеры AVR [4].

Там же приведены примеры процедур, рекомендованные производителем для этих целей. Описываемые ниже процедуры являются практически полной копией рекомендованных процедур, дополненные лишь командой автоматического увеличения адреса. Для управления памятью EEPROM используются специальные регистры ввода—вывода:

EEAR — регистр адреса;

EEDR — регистр данных;

EECR — регистр управления.

Отдельные разряды регистра управления также имеют свои собственные имена:

EEWE — бит записи;

EEMWE — бит разрешения записи;

EERE — бит чтения.

Все названия введены фирмой-производителем и правильно понимаются транслятором, если вы не забыли присоединить в начале программы файл описаний.

Порядок записи байта в EEPROM следующий. Байт данных, предназначенный для записи, должен быть помещен в регистр EEDR, а байт адреса — в регистр в EEAR. Для того, чтобы разрешить запись, необходимо установить бит EEMWE. Затем в течение четырех машинных циклов (то есть следующей же командой) нужно установить бит EEWE. Сразу же после установки бита EEWE начинается процесс записи. Этот процесс занимает довольно продолжительное время. Все это время бит EEWE остается установленным. По окончании процесса записи он сам сбрасывается в ноль. Такой многоступенчатый алгоритм придуман для предотвращения случайной записи.

Подпрограмма, реализующая описанный выше алгоритм записи байта, называется `eewr` и занимает **строки 158—166**. Байт данных, предназначенный для записи, передается в процедуру при помощи регистра `data`, а адрес ячейки, куда нужно записать данные, — через регистр `addre`. Работа подпрограммы начинается с глобального запрета всех прерываний (**строка 158**).

Это обязательное условие работы с EEPROM. Невовремя вызванное прерывание может помешать процессу записи. В данном случае запрет прерываний является избыточной мерой, так как программа построена таким образом, что при записи в EEPROM прерывания всегда запрещены.

В **строках 159, 160** расположен цикл проверки готовности EEPROM. Если бит EEWE установлен, это значит, что предыдущая операция записи еще не окончена. Поэтому в **строке 159** проверяется значение этого бита. Пока значение бита равно единице, выполняется команда безусловного перехода в **строке 160**, и проверка выполняется снова и снова.

Когда значение бита окажется равным нулю, **строка 160** будет пропущена (сработает команда `sbic` в **строке 159**), а цикл ожидания прервется. В **строке 161** происходит запись адреса из регистра `addre` в регистр EEAR. В **строке 162** в регистр EEDR записывается байт данных из регистра `data`.

В **строке 163** устанавливается бит разрешения записи. В **строке 164** — бит записи. После установки этого бита процесс записи будет

запущен. Запись будет идти своим чередом, а программа может продолжать свою работу. Главное — не менять содержимое регистров EEDR и EEAR, пока процесс записи не закончится. В **строке 165** происходит приращение содержимого регистра `addre`.

И, наконец, в **строке 166** подпрограмма завершается. Команда в **строке 165** не относится к алгоритму записи в EEPROM. Но ее применение позволяет использовать подпрограмму `eewr` для последовательной записи цепочки байтов, в чем мы и убедимся дальше.

Порядок чтения байта гораздо проще. Достаточно в регистр EEAR записать адрес ячейки, содержимое которой нужно прочитать, а затем установить бит чтения (EERE). Прочитанный байт автоматически помещается в регистр EEDR.

Подпрограмма чтения байта из EEPROM называется `eerd` и занимает **строки 167—174**. Адрес ячейки, предназначенной для чтения, передается в подпрограмму через регистр `addre`. Прочитанный байт данных подпрограмма возвращает в регистре `data`. Начинается подпрограмма чтения, как и подпрограмма записи с запрета прерываний (**строка 167**).

Так как чтению может предшествовать запись, прежде чем изменять значения регистра EEAR, нужно проверить, закончился ли процесс записи. Поэтому в **строках 168, 169** мы видим уже знакомый нам цикл ожидания готовности EEPROM. В **строке 170** в регистр EEAR записывается содержимое регистра `addre`.

В **строке 171** устанавливается бит чтения (EERE). Как только этот бит будет установлен, моментально происходит процесс чтения, и прочитанный байт данных появляется в регистре EEDR. В **строке 172** этот байт помещается в регистр `data`. В **строке 173** происходит приращение регистра адреса `addre`. Смысл этого приращения такой же, как и в предыдущем случае. Только теперь подобный прием позволяет читать из EEPROM цепочку байтов. В **строке 174** подпрограмма `eewr` завершается.

Теперь **перейдем к основной части программы**. Как уже говорилось, она занимает **строки 62—119**. И первое, что выполняет основная программа, — **цикл ожидания отпускания кнопок**. В цикле используется описанная выше подпрограмма `incod`. Она будет не только считывать код состояния клавиатуры, но и сразу же производить его сравнение.

Если вы не забыли, код состояния клавиатуры при полностью отпущенных кнопках равен 0x7F, 0x07. В **строках 62, 63** этот код записывается во вспомогательный буфер `codL+codH`. Цикл ожида-

ния отпускания кнопок расположен в **строках 64, 65**. В **строке 64** вызывается подпрограмма `incod`. Она определяет код состояния клавиатуры и сравнивает полученный код с числом, записанным в буфере `codL+codH`.

Если код состояния клавиатуры равен коду в буфере, то после выхода из подпрограммы флаг Z будет установлен. В противном случае — сброшен. Равенство кодов означает, что кнопки отпущены. Поэтому оператор условного перехода в **строке 65** проверяет значение флага Z. Пока коды разные, управление передается по метке m0, и цикл ожидания продолжается. Как только коды окажутся равными, цикл прерывается, и управление переходит к **строке 66**.

В **строке 66** начинается **цикл ожидания нажатия кнопки**. Цикл занимает **строки 66, 67** и выглядит почти так же, как цикл ожидания отпускания. Различие состоит в операторе условного перехода. Вместо `brne` (переход по условию «не равно») применяется оператор `breq` (переход по условию «равно»).

В буфере `codL+codH` по-прежнему находится код состояния клавиатуры при полностью отпущенных кнопках. Поэтому выход из данного цикла произойдет тогда, когда будет нажата любая из кнопок (S1—S10).

Как только нажатие будет обнаружено, программа переходит в следующую стадию. Полученный код состояния клавиатуры должен стать первым кодом ключевой комбинации. Но прежде чем начинать цикл ввода этой комбинации, программа выполняет две очень важные операции: В **строках 68, 69** в регистровую пару Z записывается адрес начала буфера в ОЗУ, куда будет помещаться вводимая комбинация. Регистр Z будет хранить текущий указатель этого буфера.

Вторая важная операция производится в **строке 70**. Тут обнуляется счетчик байтов, записанных в буфер. После этого начинается цикл ввода кодовой комбинации. Цикл занимает **строки 71—88**. Начинается работа цикла с формирования защитной задержки. Почему начинается с задержки, если мы только что получили первое нажатие кнопки?

А согласно алгоритму после нажатия положено формировать задержку. Для формирования защитной задержки используется подпрограмма `wait`, работающая в режиме 1. Сначала в **строке 72** в регистр `data` записывается номер режима. Затем в **строке 73** вызывается подпрограмма `wait`.

После окончания защитной задержки в **строке 74** снова производится ввод кода состояния. В **строках 75, 76** полученный код записывается в буфер. Для записи используются команды, увеличивающие значение указателя (Z). Поэтому после записи каждого очередного байта указатель передвигается в следующую позицию. Затем в **строках 77, 78** увеличивается значение счетчика принятых байтов. Так как мы записали два байта, то и значение счетчика увеличивается дважды.

В **строках 79, 80** производится оценка длины введенного кода. Если длина превысит размеры буфера, то цикл ввода кода досрочно прекращается. В **строке 79** производится сравнение текущего значения счетчика с размером буфера.

В **строке 80** находится оператор условного перехода, который передает управление по метке `m7` в случае, если длина кода превысит размер буфера. В **строках 81, 82** код состояния клавиатуры записывается в буфер codH—codL. Делается это для того, чтобы следующий введенный код состояния можно было сравнивать с текущим.

Дальше программа должна **ожидать очередное изменения кода состояния**. Но сначала нужно запустить таймер, чтобы он начал формирование защитного промежутка времени. Если в течение этого промежутка не будет нажата ни одна кнопка, то это должно послужить сигналом к выходу из цикла ввода ключевой комбинации. Запуск таймера производится при помощи подпрограммы `wait` в режиме номер два. В **строке 83** в регистр `data` записывается номер режима, а в **строке 84** вызывается сама подпрограмма.

В **строках 85—88** организован комбинированный цикл ожидания. В теле цикла происходит сразу несколько операций. Во-первых, вводится новое значение кода состояния клавиатуры (**строка 85**). В процессе ввода новый код сравнивается со старым, который хранится в буфере codH—codL. Если коды не равны, то оператор условного перехода в **строке 86** передает управление по метке `m3`, где происходит формирование защитной задержки, затем повторное считывание и запись кода в буфер и так далее.

Если новое значение кода равно старому, комбинированный цикл продолжается. В **строке 87** производится проверка флага задержки `flz`. Если флаг равен нулю, это значит, что защитный промежуток времени еще не закончился. В этом случае оператор условного перехода в **строке 88** передает управление по метке `m6`, и комбинированный цикл продолжается сначала. Если значение флага `flz` равно единице, то цикл завершается, и управление переходит к **строке 89**.

В **строках 89, 90** происходит проверка переключателя режимов работы (S11). В зависимости от состояния этого переключателя полученная только что кодовая комбинация либо записывается в EEPROM (режим «Запись»), либо поступает в процедуру проверки (режим «Работа»). Команда `sbic` в **строке 89** проверяет значение седьмого бита регистра PINB.

Если бит равен нулю (контакты тумблера замкнуты), то **строка 90** не выполняется, и управление переходит к **строке 91**, где начинается процедура записи в EEPROM. Если значение разряда равно единице (контакты тумблера не замкнуты), оператор условного перехода в **строке 90** передает управление по метке `m9` на начало процедуры сравнения.

Процедура записи ключевой комбинации в EEPROM

Эта процедура занимает **строки 91—101**. К началу этой процедуры кодовая комбинация уже находится в буфере ОЗУ. Длина комбинации содержится в переменной `count`. Нам остается только записать все это в EEPROM.

В **строках 91—93** в EEPROM записывается длина комбинации. В качестве адреса для записи используется метка `klen`. Эта метка указывает на ячейку, которая специально зарезервирована для этой цели (см. **строку 20**). Для записи байта в EEPROM используется подпрограмма `eewr`.

В **строке 91** длина комбинации помещается в регистр `data`. В **строке 92** адрес помещается в регистр `addre`. Затем вызывается подпрограмма `eewr` (**строка 93**).

В **строках 97—100** расположен цикл записи всех байтов ключевой комбинации. Перед началом цикла в регистр `addre` записывается адрес первой ячейки буфера-приемника, находящегося в EEPROM (**строка 94**). А в регистровую пару Z записывается адрес первой ячейки буфера-источника, находящегося в ОЗУ (**строки 95, 96**).

В процессе записи ключевой комбинации регистр `count` используется для подсчета записанных байтов. В начале, как уже говорилось, он содержит длину комбинации. При записи каждого байта содержимое `count` уменьшается. Когда оно окажется равным нулю, цикл записи прекращается.

Цикл начинается с того, что очередной байт ключевой комбинации, находящейся в ОЗУ, помещается в регистр data (**строка 97**). Напомню, что адрес уже находится в регистре addre. В **строке 98** вызывается подпрограмма, которая записывает байт в EEPROM. Та же подпрограмма увеличивает значение addre на единицу. В **строке 99** уменьшается значение регистра count.

Проверку содержимого count на равенство нулю производит оператор brne в **строке 100**. Если содержимое не равно нулю, то оператор передает управление на метку m8, и цикл записи ключевой комбинации продолжается. В противном случае цикл завершается, и управление переходит к **строке 101**. То есть к процедуре открывания замка.

Теперь разберемся, зачем после записи кода вызывается процедура открывания замка. Это сделано для удобства. После ввода кодовой комбинации необходимо выдержать паузу в 1 с для того, чтобы введенная комбинация записалась в EEPROM. Для того, чтобы точно знать, когда заканчивается эта пауза, используется срабатывание замка. Как только щелкнет соленоид, можно считать, что ввод закончен.

Процедура проверки кода

Эта процедура занимает **строки 102—114**. Процедура проверки во многом похожа на процедуру записи. Для чтения байта из EEPROM используется подпрограмма eerd. Перед вызовом этой подпрограммы адрес ячейки, откуда будет прочитана информация, записывается в регистр addre. Прочитанное из EEPROM значение возвращается в регистре data.

В **строках 102—105** происходит считывание длины последовательности из EEPROM и сравнение ее с длиной новой последовательности. Сначала в **строке 102** адрес ячейки, где хранится длина кода, записывается в addre. Затем вызывается подпрограмма eerd (**строка 103**).

В **строке 104** сравнивается полученное из EEPROM значение (регистр data) и новое значение длины кода (регистр count). В случае неравенства этих двух величин оператор условного перехода в **строке 105** передает управление в начало программы. В этом случае дальнейшая проверка больше не производится. Дверь остается закрытой.

Если длина кода оказалась правильной, начинается цикл побайтной проверки кодов. Сначала в регистр `addre` записывается начальный адрес буфера в EEPROM (**строка 106**). В регистровую пару Z записывается начальный адрес буфера в ОЗУ (**строки 107, 108**). Сам цикл сравнения занимает **строки 109—114**.

В **строке 109** вызывается подпрограмма, которая читает очередной байт из EEPROM. Байт помещается в регистр `data`. В **строке 110** читается байт из ОЗУ. Этот байт помещается в регистр `temp`. В **строке 111** эти два байта сравниваются. Если байты не равны, оператор условного перехода в **строке 112** прерывает процесс сравнения и передает управление на начало программы. То есть дверь и в этом случае остается закрытой.

Если байты равны, выполняется уменьшение содержимого счетчика `count` (**строка 113**). Если после уменьшения содержимое `count` еще не достигло нуля, управление передается по метке `m10` (оператор `brne` в **строке 114**), и цикл сравнения продолжается. Когда содержимое `count` окажется равным нулю, процесс сравнения заканчивается. Это означает, что все байты обеих версий ключевой комбинации оказались равны. Поэтому программа плавно переходит к процедуре открывания замка (к **строке 115**).

Процедура открывания замка

Эта процедура занимает **строки 115—119**. Процедура очень проста. Для открывания замка на четвертый разряд порта PB подается единичный сигнал, который открывает транзистор ключа VT1 (см. **рис. 1.17**). Реле срабатывает, и замок открывается. Подав открывающий сигнал, программа выдерживает паузу, а затем сигнал снимает. После этого замок закрывается. Длительность паузы равна одной секунде. Этого времени достаточно, чтобы открыть дверь.

Подача открывающего сигнала на выход осуществляется в **строке 115**. В **строках 116, 117** производится вызов процедуры задержки. При этом выбирается режим номер 3. Сначала в регистр `data` помещается код режима задержки (**строка 116**). Затем вызывается подпрограмма `wait` (**строка 117**). В **строке 118** снимается сигнал открывания двери. В **строке 119** процедура открывания замка завершается. Оператор безусловного перехода, находящийся в этой строке, передает управление на начало программы. И весь процесс начинается сначала.

Программа на языке СИ

Возможный вариант программы кодового замка на языке СИ приведен в **листинге 1.20**. Программа реализует тот же самый алгоритм, что и приведенная выше программа на Ассемблере. В тексте этой программы были использованы несколько новых для нас элементов языка СИ. Рассмотрим их по порядку.

#define

Директива присвоения символьного имени любой константе. Это классический элемент языка СИ, который поддерживается любой версией языка. Директива имеет два параметра. Первый параметр — имя константы. Второй параметр — ее значение. В **строке 2** программы (**листинг 1.20**) числовой константе 0x77F присваивается имя `klfree`. После этого в любом месте программы, где нужно использовать число 0x77F, его можно заменить именем `klfree`.

В качестве значения константы может выступать не только число, но и любая комбинация чисел и букв. И даже комбинация, состоящая только из букв. Применение именованных констант делает программу более наглядной. Кроме того, изменять значение такой константы становится удобнее. Достаточно заменить значение константы в одном только месте (в строке описания). И сразу же новое значение будет учтено по всей программе.

#pragma

Директива, задающая специальные команды для компилятора. В качестве параметра в директиве указывается задаваемая команда. Ниже приведен ряд примеров использования этой директивы.

Включение/отключение сообщений об ошибках.

#pragma warn– // Отключает предупреждающие сообщения.

#pragma warn+ // Включает предупреждающие сообщения.

Включение/отключение оптимизации результирующего кода.

#pragma opt– // Отключает оптимизацию.

#pragma opt+ // Включает оптимизацию.

Включение/отключение оптимизации по минимальному размеру результирующего кода.

#pragma optsize– // Отключает оптимизацию по размеру.

#pragma optsize+ // Включает оптимизацию по размеру.

И так далее. Полный список всех команд, передаваемых посредством этой директивы, вы можете найти в файле помощи программы CodeVision в разделе «Препроцессор» («The Preprocessor»).

eeprom

Управляющее слово, используемое при описании переменных (массивов), которое указывает транслятору, что данная переменная (массив) будет располагаться в энергонезависимой памяти данных (EEPROM). **Например**, в **строке 9** программы находится описание переменной `klen`, предназначенной для хранения длины кодовой комбинации, а в **строке 10** описывается массив, в котором будет храниться сама комбинация. И переменная, и массив размещаются в EEPROM.

return

Команда возврата значения. Если функция языка СИ должна возвращать значение, последней командой в теле этой функции должна быть команда `return`. В качестве параметра этой команды указывается возвращаемое значение.

Примером может служить функция `incod()`, занимающая в нашей программе (**листинг 1.20**) **строки 16—26**. Функция производит определение кода состояния клавиатуры с использованием процедуры антидребезга. По результатам своей работы функция должна возвращать код состояния клавиатуры. Поэтому последняя команда в теле функции (**строка 26**) — это команда `return`. В качестве параметра эта команда использует переменную `cod1`, которая и содержит сформированный код состояния.

Кроме новых команд, в тексте программы (**листинг 1.20**) используется один, пока еще не знакомый нам интересный прием. Посмотрите, пожалуйста, на **строку 54** программы. В этой строке записано выражение, которое присваивает элементу массива `bufr` значение переменной `codS`. Однако в качестве номера элемента массива используется не просто переменная `ii`, а выражение `ii++`. Это и есть еще одна оригинальная особенность языка СИ.

Язык СИ допускает одновременно использовать переменную в любом выражении и изменять ее значение. Так, при вычислении выражения `bufr[ii++]=codS` сначала элементу массива с номером `ii` присваивается значение `codS`, а затем значение переменной `ii` увеличивается на единицу. Такой же прием допускается при использовании переменных в качестве параметров функций или в составе любых других выражений. Кроме команды увеличения, можно использовать команду уменьшения, а также менять порядок вычислений. Вот несколько примеров таких выражений:

```
a=MyBuffer[++ii]; b=MyFunction(ii--); c=85+(--ii)/2;
```

В любом случае, при использовании данного приема выполняются следующие правила:

ii++ означает: использовать значение, а затем увеличить его на единицу.

++ii означает: увеличить значение на единицу, а затем использовать его.

ii-- означает: использовать значение, а затем уменьшить его на единицу.

--ii означает: уменьшить значение на единицу, а затем использовать его.

Естественно, вместо переменной `ii` может использоваться любая другая переменная.

Описание программы (листинг 1.20)

Кардинальным **отличием** программы на языке СИ является тот факт, что все шестнадцатиразрядные значения, используемые в программе, теперь не нужно разбивать на отдельные байты. Для хранения каждой такой величины программа использует либо переменную, либо константу соответствующего типа.

Например, для хранения разных вариантов кода состояния клавиатуры в программе используется несколько переменных:

`cod0` — **строка 17**;

`cod1` — **строка 18**;

`codS` — **строка 37**.

Листинг 1.20

```
/*****************************************************
Project : Пример 10
Version : 1
Date    : 07.03.2006
Author  : Belov
Company : Home
Comments:
Кодовый замок

Chip type           : ATtiny2313
Clock frequency     : 4,000000 MHz
Memory model        : Tiny
Data Stack size     : 32
*****************************************************/

#include <tiny2313.h>

#define klfree 0x77F            // Код состояния при полностью отпущенных кнопках
#define kzad 3000               // Код задержки при сканировании
#define kandr 20                // Константа антидребезга
#define bsize 30                // Размер буфера для хранения кода

unsigned char flz;                  // Флаг задержки
unsigned int bufr[bsize];           // Буфер в ОЗУ для хранения кода
#pragma warn-
eeprom unsigned char klen;          // Ячейка для хранения длины кода
eeprom unsigned int bufe[bsize];    // Буфер в EEPROM для хранения кода
#pragma warn-

// Прерывание по переполнению Таймера 1
interrupt [TIM1_OVF] void timer1_ovf_isr(void)
{
  flz=1;
}

// Прерывание по совпадению в канале А Таймера 1
interrupt [TIM1_COMPA] void timer1_compa_isr(void)
{
  flz=1;
}

// Функция опроса клавиатуры и антидребезга
unsigned int incod (void)
{                                   // Создаем локальные переменные
  unsigned int cod0=0;              // Вспомогательная переменная
  unsigned int cod1;                // Еще одна вспомогательная переменная
  unsigned char k;                  // Параметр цикла антидребезга

  for (k=0; k<kandr; k++)           // Цикл антидребезга
  {
      cod1=PINB&0x7;                // Формируем первый байт кода
      cod1=(cod1<<8)+(PIND&0x7F);   // Формируем полный код состояния клавиатуры
      if (cod0!=cod1)               // Сравниваем с первоначальным кодом
      {
            k=0;                  // Если не равны, сбрасываем счетчик
            cod0=cod1;            // Новое значение первоначального кода
      }
  }
  return cod1;
}

// Процедура формирования задержки
void wait (unsigned char kodz)
{
  if (kodz==1) TIMSK=0x40;        // Выбор маски прерываний по таймеру
  else TIMSK=0x80;
  TCNT1=0;                        // Обнуление таймера
  flz=0;                          // Сброс флага задержки
  #asm("sei");                    // Разрешаем прерывания
```

```
    if (kodz!=2) while(flz==0);     // Цикл задержки
}

// Основная функция
void main(void)
{
    unsigned char ii;            // Указатель массива
    unsigned char i;             // Вспомогательный указатель
    unsigned int codS;           // Старый код

    PORTB=0xE7;                  // Порт B
    DDRB=0x18;

    PORTD=0x7F;                  // Порт D
    DDRD=0x00;

    TCCR1A=0x00;                 // Таймер/Счетчик 1
    TCCR1B=0x03;
    TCNT1=0;                     // Обнуление счетного регистра
    OCR1A=kzad;                  // Инициализация регистра совпадения

    ACSR=0x80;                   // Аналоговый компаратор

    while (1)
    {
    m1:        while (incod() != klfree);     // Ожидание отпускания кнопок
               while (incod() == klfree);     // Ожидание нажатия кнопок
               ii=0;
    m2:        #asm("cli");                   // Запрещаем прерывания
               wait(1);                       // Задержка 1-го типа
               codS=incod();                  // Ввод кода и запись, как старого
               bufr[ii++]=codS;               // Запись очередного кода в буфер
               if (ii>=bsize) goto m4;        // Проверка конца буфера

               wait(2),                       // Задержка 2-го типа
    m3:        if (incod() != codS) goto m2;  // Проверка, не изменилось ли состояние
               if (flz==0) goto m3;           // Проверка флага окончания задержки

    m4:        if (PINB.7==1) goto comp,      // Проверка переключателя режимов

    //----------------------------- Запись кода в EEPROM

               klen=ii;                                  // Запись длины кода
               for (i=0; i<ii; i++) bufe[i]=bufr[i];     // Запись всех байтов кода
               goto zamok;                               // К процедуре открывания замка

    //----------------------------- Проверка кода

    comp:      if (klen!=ii) goto m1;                    // Проверка длины кода
               for (i=0; i<ii; i++)  if (bufe[i]!=bufr[i]) goto m1;    // Проверка кода

    //----------------------------- Открывание замка

    zamok:     PORTB.4=1;        // Открываем замок
               wait(3);          // Задержка 3-го типа
               PORTB.4=0;        // Закрываем замок
    }
}
```

И все они имеют тип `unsigned char`. При этом сам этот код представляет собой шестнадцатиразрядное двоичное число, старшие восемь разрядов соответствуют содержимому порта PB, а младшие восемь разрядов — содержимому порта PD (на все это наложена маска).

Начинается наша программа, как и все предыдущие, с присоединения библиотечного файла (**строка 1**). Далее идет блок описания констант (**строки 2—5**). Первая константа имеет имя `klfree` и значение 0x77F. Это значение представляет собой код состояния клавиатуры при отпущенных кнопках. Константа используется в операциях сравнения.

Следующие три константы: `kzad` (код задержки), `kandr` (константа антидребезга) и `bsize` (размер буфера для кодовой комбинации) по своему назначению аналогичны соответствующим константам в программе на Ассемблере. У них даже значения одинаковы. Отличие только в значении константы `bsize`. В нашем случае она равна не 60, а 30.

Почему буфер стал вдвое короче? Дело в том, что на Ассемблере буфер представлял собой набор ячеек памяти размером в один байт каждая. Любое новое значение записывалось в буфер в виде двух отдельных байтов. В программе на СИ в качестве буфера будет использоваться массив, состоящий из шестнадцатиразрядных элементов. Каждое значение в такой буфер заносится как один отдельный элемент.

В **строках 6—11** находится блок описания переменных и массивов. Все переменные и массивы имеют свои аналоги в программе на Ассемблере. В **строке 6** описывается переменная `flz`, которая используется как флаг задержки. В **строке 7** описывается массив для оперативного хранения ключевой комбинации. Все значения этого массива будут размещены в ОЗУ (буфер в ОЗУ). Длина массива выбирается равной `bsize`.

В **строках 9, 10** описываются переменная и массив, которые будут храниться в EEPROM. Переменная `klen` предназначена для хранения длины кодовой комбинации. Массив `bufe` предназначен для хранения самой этой комбинации.

При описании переменных и массивов, которые будут размещаться в EEPROM, данная версия языка СИ требует обязательной их инициализации. То есть требует указать значение всех элементов по умолчанию. Указанные таким образом значения в процессе «прошивки» микроконтроллера попадают непосредственно в EEPROM.

Если соблюдать указанные выше требования, то **строки 9 и 10** нашей программы должны выглядеть примерно так:

```
eeprom unsigned char klen=0x4;
```

```
eeprom unsigned int bufe[bsize]={0x76F, 0x77F, 0x7FE, 0x77F};
```

Если мы воспользуемся данной редакцией команд описания, то мы получим программу электронного кодового замка. В ней уже на стадии изготовления заложена некоторая начальная ключевая кодовая комбинация, которую, впрочем, в любой момент владелец замка может изменить на новую.

Но мне интересно показать, что можно сделать, если закладывать код заранее нежелательно. Если вы не стали указывать начальные значения для переменной и массива, то это не является критической ошибкой. Программа будет успешно оттранслирована, а результирующий код полностью работоспособен. Единственное неудобство — сообщение о некритичной ошибке. По-английски оно называется «Warning» (предупреждение). Оно будет возникать каждый раз при трансляции программы. Можно, конечно, просто не обращать на него внимание.

Однако более правильно будет временно отключить сообщение при помощи директивы `#pragma` и команды `warn`, как это и сделано в программе на **листинге 1.20**. В **строке 8** вывод предупреждений отключается, а в **строке 11** включается снова. Отключать предупреждения навсегда не рекомендуется. Так можно пропустить другие, более важные предупреждения.

Далее в программе начинается **описание всех составляющих ее функций**. Данная программа состоит из пяти функций:

- две функции обработки прерываний (**строки 12, 13** и **14, 15**);
- функция ввода кода состояния клавиатуры (**строки 16—26**);
- функция формирования задержки (**строки 27—33**);
- главная функция программы (**строки 34—67**).

Рассмотрение данной программы удобно начинать с главной функции `main`.

Функция `main` начинается с описания локальных переменных (**строки 35—37**). Кроме переменной `codS`, предназначенной для временного хранения кода состояния клавиатуры, здесь определяются еще две вспомогательные переменные с именами `i` и `ii`. После описания переменных начинается **блок инициализации (строки 38—46)**. Блок инициализации данной программы по выполняемым

действиям полностью повторяет аналогичный блок в программе на Ассемблере.

Эти действия сводятся к настройке портов ввода—вывода, таймера и компаратора. При настройке таймера не только выбирается его режим работы, но и обнуляется значение счетного регистра TCNT1 (**строка 44**), а в регистр совпадения OCR1A записывается код задержки kzad (**строка 45**).

Основной цикл программы занимает **строки 47—67**. Рассмотрим подробнее его работу. В **строке 48** находится цикл ожидания отпускания кнопок. Он представляет собой пустой цикл `while`. Тело цикла полностью отсутствует. За ненадобностью не поставлены даже фигурные скобки. Весь цикл состоит лишь из оператора `while` и выражения в круглых скобках, определяющего условие продолжения этого цикла.

Условие простое: цикл выполняется все время, пока код состояния клавиатуры и константа `klfree` не равны между собой. Значение константы равно коду состояния клавиатуры при всех отпущенных кнопках. Для определения кода состояния клавиатуры используется функция `incod()`.Функция `incod()` выполняет те же самые действия, что одноименная процедура из программы на Ассемблере. То есть считывает состояние портов, накладывает маски и применяет при этом антидребезговый алгоритм. Подробнее работу функции мы рассмотрим в конце этого раздела.

Как только все кнопки будут отпущены, цикл в **строке 48** завершается, и программа переходит к **строке 49**, в которой находится цикл ожидания нажатия любой кнопки. Этот цикл очень похож на предыдущий. Изменилось только условие. Знак != (не равно) заменен на == (равно). Как только будет нажата любая кнопка, цикл в **строке 49** заканчивается, и управление переходит к **строке 50**.

Теперь пора начинать **цикл ввода ключевой кодовой комбинации**. Но сначала нужно обнулить переменную `ii`, которая будет использоваться как счетчик принятых кодов и указатель текущего элемента в буфере `bufr`. Обнуление выполняется в **строке 50**. Цикл ввода кодовой комбинации занимает **строки 51—58**. Начинается цикл с глобального запрета всех прерываний (**строка 51**). Затем формируется защитная пауза. Для формирования паузы используется функция `wait()`.

Действие этой функции полностью аналогично действию одноименной процедуры из программы на Ассемблере. В **строке 52** формируется задержка первого вида (48 мс), то есть вызывается

функция wait() с параметром, равным единице. По окончании задержки (**строка 53**) программа повторно считывает код состояния клавиатуры и записывает его в переменную codS.

В **строке 54** считанный код записывается в буфер ОЗУ (bufr). Одновременно указатель буфера увеличивается на единицу. В **строке 55** проверяется условие переполнения буфера. Такая проверка принудительно завершает работу цикла при попытке ввода слишком длинной кодовой комбинации. Если значение ii превысило величину константы bsize, значит, буфер уже полностью заполнен. В этом случае управление передается по метке m4, то есть на конец цикла.

Если значение ii не достигло конца буфера, то перехода не происходит, и управление переходит к **строке 56**. В этом месте запускается процесс формирования контрольного промежутка времени. Для запуска этого процесса используется функция wait(). В данном случае она формирует задержку второго типа, поэтому вызывается с параметром, равным двум.

Так же, как и в программе на Ассемблере, задержка второго типа лишь производит все настройки таймера, но не выполняет цикл ожидания. Комбинированный цикл ожидания находится вне функции задержки, а точнее в **строках 57, 58**. В **строке 57** считывается новый код состояния клавиатуры и сравнивается со старым, который хранится в переменной codS.

Если коды не равны (состояние клавиатуры изменилось), комбинированный цикл ожидания прерывается, и управление передается по метке m2. То есть на начало цикла ввода кодовой комбинации. А уже там, в начале цикла, снова формируется защитная задержка, и очередной код состояния помещается в буфер.

Если при проверке в **строке 57** старый и новый коды оказались все же равны между собой, перехода не происходит, и выполняется **строка 58**. В этой строке проверяется значение флага flz. Если контрольный промежуток времени еще не истек, то значение флага равно нулю, и управление передается по метке m3. Комбинированный цикл продолжается.

Если же контрольный промежуток времени уже закончится, значение флага flz равно единице. Поэтому перехода не происходит, и управление переходит к **строке 59**. На этом и комбинированный цикл ожидания, и цикл ввода кодовой комбинации заканчиваются.

В **строке 59** проверяется состояние тумблера s11. В зависимости от этого состояния выполняется либо запись только что принятой

кодовой комбинации в EEPROM, либо извлечение из EEPROM и сравнение двух кодовых комбинаций. Для проверки состояния переключателя оценивается значение седьмого разряда порта PB. Если контакты переключателя разомкнуты (PINB.7 равен единице), то управление передается по метке `comp` (к процедуре сравнения кодов). В противном случае выполняется процедура записи.

Процедура записи кода в EEPROM занимает **строки 60—62**. В **строке 60** длина кодовой комбинации записывается в переменную `klen`. Так как при описании переменной `klen` (см. **строку 9**) ее местом расположения выбран EEPROM, то записанное в переменную значение автоматически туда и попадает. Язык СИ сам выполняет все необходимые для этого процедуры. Как видите, в языке СИ запись в EEPROM происходит гораздо проще, чем на Ассемблере.

В **строке 61** находится цикл, который производит запись в EEPROM самой кодовой комбинации. Цикл просто по очереди записывает каждый элемент массива `bufr` в соответствующий элемент массива `bufe`. А `bufe` целиком находится в EEROM. В качестве параметра цикла используется переменная `i`. По ходу работы цикла значение этой переменной меняется от нуля до `ii`. То есть перебираются номера всех элементов кодовой комбинации (`ii` равно ее длине). Тело цикла составляет всего одно выражение. Это выражение записывает значение очередного элемента буфера `bufr` в буфер `bufe`.

Причем в качестве указателя для обоих массивов используется одна и та же переменная. Поэтому в буфере `bufe` элементы попадают в те же позиции, какие они занимали в буфере `bufr`. По окончании цикла записи управление переходит к **строке 62**. В этой строке находится оператор безусловного перехода, который передает управление по метке `zamok`. То есть к процедуре открывания замка. Щелчок механизма замка оповещает об окончании процесса записи.

Процедура проверки занимает **строки 63, 64**. Напомним, что к началу этой процедуры переменная `ii` содержит длину только что введенной кодовой комбинации, а буфер `bufr` — саму эту комбинацию. Сначала, в **строке 63**, сравнивается значение переменной `klen` (длина, записанная ранее в EEPROM) и значение переменной `ii`. Язык СИ сам извлекает значение `klen` из EEPROM, используя все необходимые процедуры.

Если в результате проверки эти две длины окажутся не равными, то управление передается по метке m1. То есть к началу всей программы. Дальнейшее сравнение кодов не производится, и замок не открывается. Если оба значения одинаковы, то программа переходит к сравнению кодовых комбинаций. Цикл, производящий это

сравнение, находится в **строке 64**. Этот цикл в качестве параметра тоже использует переменную `i`.

В процессе работы цикла значение этой переменной также меняется от нуля до `ii`. В теле цикла выполняется оператор сравнения `if`. Этот оператор сравнивает значения элементов двух массивов, один из которых (`bufr`) находится в EEPROM, а второй (`bufe`) расположен в ОЗУ. При первом же несовпадении кодов оператор безусловного перехода передает управление по метке m1. В этом случае цикл проверки досрочно прерывается, и замок не открывается. Если в процессе работы цикла проверки все коды оказались одинаковыми, то цикл завершается нормальным образом, и управление переходит к **строке 65**. То есть к процедуре открывания замка.

Процедура открывания замка очень проста. Она занимает **строки 65—67**. **В строке 65** подается команда, открывающая механизм замка (в четвертый разряд порта PB записывается единица). Затем вызывается задержка третьего типа (**строка 66**). По окончании задержки открывающий сигнал снимается (**строка 67**). С окончанием процедуры открывания замка заканчивается тело основного цикла программы. Так как основной цикл бесконечный, то управление передается на его начало. То есть на **строку 48**. Работа программы начинается сначала.

Теперь вернемся **к вспомогательным функциям программы**, которые мы пропустили вначале этого описания. Начнем по порядку. **В строках 12, 13** и **14, 15** размещены две разные по названию, но одинаковые по содержанию функции. Первая из них является процедурой обработки прерывания по переполнению таймера/счетчика1. А вторая — процедурой обработки прерывания по совпадению в канале А того же таймера.

В программе на Ассемблере оба вида прерываний вызывали одну и ту же общую процедуру. Данная версия языка СИ не позволяет использовать одну и ту же функцию в качестве процедуры обработки двух разных видов прерываний. В теле каждой из функций имеется всего одна строка. В этой строке присваивается единица переменной `flz`. То же самое делает процедура обработки прерывания в программе на Ассемблере.

Строки 16—26 занимает функция ввода состояния клавиатуры. Алгоритм работы этой функции немного отличается от алгоритма работы аналогичной процедуры на Ассемблере. Функция `incod()` в программе на языке СИ производит лишь считывание содержимого портов, наложение маски и антидребезговый алгоритм.

Операция сравнения в теле данной функции не выполняется. Так как функция `incod()` должна возвращать код состояния клавиатуры, тип возвращаемого значения определен как `unsigned int` (см. **строку 16**). Функция не имеет параметров, на что указывает слово `void`.

Рассмотрим подробнее, как работает функция `incod()`. **В строках 17—19** производится описание локальных переменных. Переменные `cod0` и `cod1` используются для хранения промежуточных значений кода состояния клавиатуры. Причем переменная `cod1` используется для хранения нового значения кода, а переменная `cod0` — для хранения старого значения (не путайте с буфером `codS` основной программы). При описании переменной `cod0` (**строка 17**) одновременно производится ее инициализация (присваивается нулевое значение). Этот ноль необходим для правильного начала цикла антидребезга. Для того чтобы при первом сравнении новый код не был равен старому. Еще одна переменная с именем `k` используется в качестве параметра цикла антидребезга.

Основу функции `incod()` составляет цикл антидребезга (**строки 20—25**). По сути, тело функции состоит только из этого цикла. Задача цикла — ввести код состояния клавиатуры заданное количество раз (определяется константой `kandr`). В данном случае используется стандартный цикл `for` (см. **строку 20**).

Цикл выполнится полностью (нужное количество раз) в том случае, если за время его работы код состояния клавиатуры не изменится. Если в процессе выполнения цикла состояние кнопок изменяется, то специальные команды внутри цикла перезапускают его работу сначала. Вычисление кода состояния производится в **строках 21** и **22**. Оператор `if` (**строка 23**) сравнивает старое и новое значения кодов. Если эти значения не равны, выполняются команды перезапуска цикла (**строки 24, 25**). По окончании работы цикла антидребезга команда `return` определяет возвращаемое значение (**строка 26**). На этом функция завершается.

Разберемся с отдельными элементами цикла антидребезга подробнее. И начнем со **строк 21** и **22**, где, как уже говорилось, происходит формирование кода состояния клавиатуры. Суть производимых вычислений наглядно проиллюстрирована на **рис. 1.18**. Источником информации для этих вычислений является содержимое регистров PINB и PIND, которые, как известно, непосредственно подключены к выводам портов PB и PD.

На содержимое обоих портов накладываются соответствующие маски, а затем все это объединяется в одно шестнадцатиразрядное

число и помещается в регистр `cod1`. Выражение в **строке 21** соответствует первому этапу на **рис. 1.18**. В правой части выражения выполняется операция логического умножения (операция «И») между содержимым порта PB и маской 0x07. Результат выражения записывается в переменную `cod1`.

Выражение в **строке 22** объединяет этапы 2 и 3. **Правая часть этого выражения** представляет собой сумму двух слагаемых. Первое слагаемое представляет собой операцию логического сдвига содержимого переменной `cod1` на восемь разрядов влево. Этот сдвиг соответствует этапу номер 2 на рисунке. В результате сдвига восемь младших разрядов числа становятся старшими.

Второе слагаемое — это еще одна операция логического умножения. На этот раз умножается содержимое порта PD и константа 0x7F, представляющая собой маску для этого порта. Результатом сложения двух этих слагаемых является искомый код состояния клавиатуры, который записывается в переменную `cod1`. Вычисление второго слагаемого и сложение их обоих и соответствует этапу номер 3 на **рис. 1.18**. Символом «X» на рисунке обозначаются разряды, значение которых не определено. Разряды, значение которых зависит от той либо иной кнопки клавиатуры, обозначается названием этой кнопки. Остальные разряды равны либо «0», либо «1».

Итак, в результате выполнения описанных выше операций переменная `cod1` содержит новое значение кода состояния клавиатуры. В **строке 23** этот код сравнивается со старым значением, которое хранится в переменной `cod0`. Если коды не равны, то выполняются команды перезапуска цикла (**строки 24, 25**). В **строке 24** переменной k (параметру цикла) присваивается нулевое значение. В **строке 25** значение переменной `cod1` записывается в переменную `cod0`. Теперь новое, только что полученное значение кода становится старым.

В **строках 27—33** расположена **функция формирования задержки**. Алгоритм работы и этой функции повторяет алгоритм работы аналогичной процедуры на Ассемблере. Функция не возвращает никаких значений, но зато имеет входной параметр: **код вида задержки**.

Как и процедура на Ассемблере, функция `wait()` формирует три вида задержки. Параметр, определяющий номер задержки, имеет имя `kodz` и тип `unsigned char`. Работа функции начинается с определения значения маски прерываний. Для этого в **строках 28, 29** оценивается значение переменной `kodz`. Если значение `kodz`

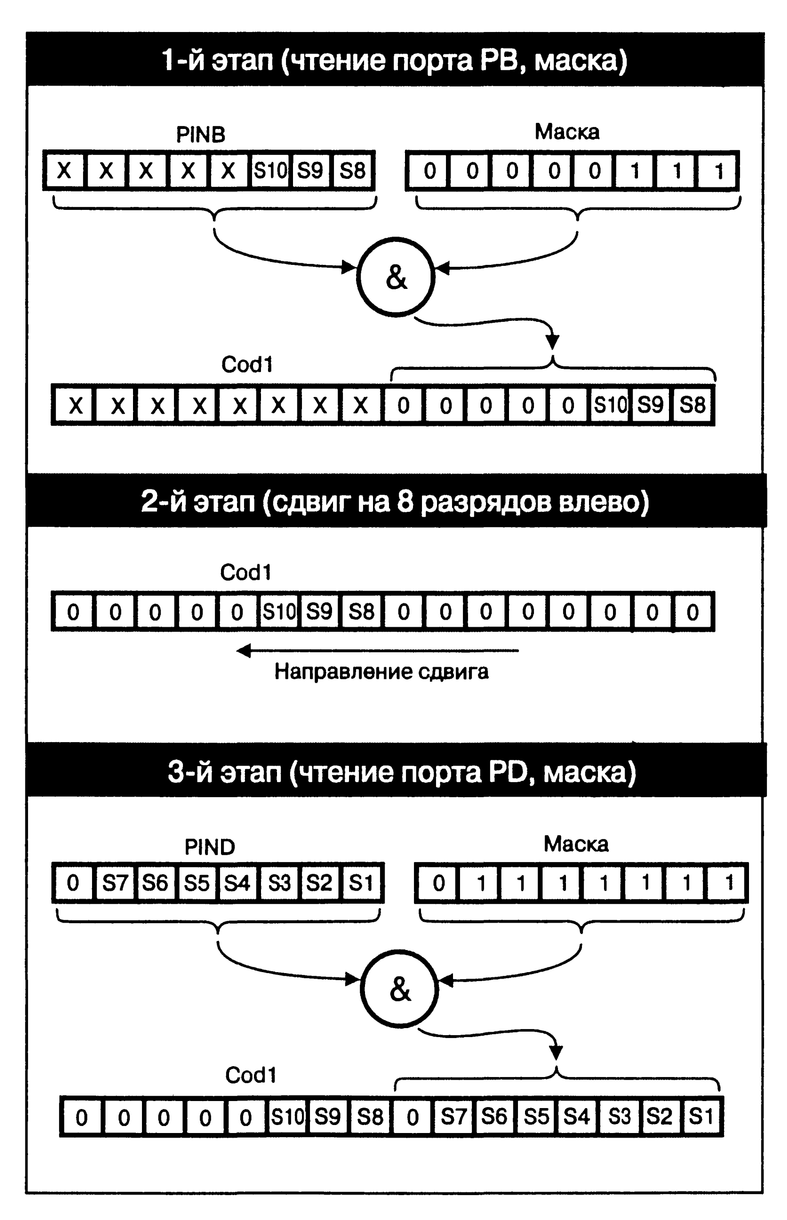


Рис. 1.18. *Формирование кода состояния клавиатуры*

равно 1, то в регистр маски прерываний TIMSK записывается код 0x40. Этот код разрешает прерывание по совпадению в канале А.

Если kodz не равен единице, то регистру TIMSK присваивается значение 0x80 (прерывание по переполнению). Таким образом, в режиме 1 будет работать прерывание по совпадению в канале А. При этом формируется задержка длительностью 48 мс. В остальных режимах (2, 3) используется прерывание по переполнению таймера. В этом случае формируется задержка длительностью в одну секунду.

В **строке 30** обнуляется значение счетного регистр таймера. С момента обнуления таймера начинается формирование заданного временного интервала. В **строке 31** обнуляется значение флага задержки. В **строке 32** выполняется команда глобального разрешения прерываний. На этом настройка таймера и системы прерываний заканчиваются. Теперь остается лишь организовать **цикл ожидания**.

Цикл ожидания, предназначенный для работы в режимах 1 и 3, организован в **строке 33**. Это пустой цикл, организованный при помощи оператора while. В качестве условия продолжения цикла выбрано равенство флага flz нулю. То есть пока flz равен нулю, цикл ожидания будет продолжаться. А закончится он в тот момент, когда процедура обработки прерывания изменит значение флага flz на единичное.

В режиме 2 используется другой цикл ожидания, который находится вне функции wait(). Поэтому в **строке 33**, кроме цикла ожидания, имеется оператор сравнения if. Он проверяет значение переменной kodz. Благодаря оператору сравнения, цикл ожидания в **строке 33** выполняется только в том случае, когда kodz не равен двум.

1.12. Кодовый замок с музыкальным звонком

Ну и в заключении приведу пример, как можно объединить вместе две разные задачи. Посмотрим, как можно соединить описанный выше кодовый замок и музыкальную шкатулку. Музыкальная шкатулка с успехом может выполнять функцию дверного звонка. Нужно лишь немного изменить алгоритм запуска музыкальной программы и заставить шкатулку играть разные мелодии при нажатии одной кнопки.

Постановка задачи

Естественно, далеко не любые две задачи можно объединить в одном устройстве. Но данные две задачи прекрасно объединяются. Если немного доработать музыкальную шкатулку, то легко заставить ее работать от одной кнопки. Менять мелодии можно при каждом очередном нажатии этой кнопки.

Звучание мелодий должно начинаться при нажатии кнопки и продолжаться до ее отпускания. При следующем нажатии должна звучать другая мелодия. Доработанная таким образом программа будет использовать всего две линии порта. В схеме электронного замка как раз есть две свободные линии.

С точки зрения объединения двух программ тоже нет никаких проблем. Так как обе эти программы вполне могут работать по отдельности. Действительно, кто же будет одновременно звонить в звонок и набирать кодовую комбинацию?

Итак, мы можем **сформулировать последнюю нашу задачу** следующим образом:

«Объединить два разработанных ранее устройства: музыкальную шкатулку (пример 9) и электронный кодовый замок (пример 10) в одно универсальное устройство, обладающее обеими этими функциями. Устройство должно иметь десять кнопок набора кода, тумблер выбора режима, выходной каскад для управления электромагнитом замка, а также кнопку звонка и выходной каскад звука».

Алгоритм

Итак, нам нужно объединить два алгоритма. Как уже говорилось выше, эти два алгоритма не должны работать одновременно. Поэтому каждый алгоритм останется практически без изменений.

Но их нужно все же соединить вместе. Как может выглядеть это соединение?

Для соединения алгоритмов достаточно объединить их процедуру ожидания. Такая процедура есть и в программе музыкальной шкатулки, и в программе замка. В первом случае такая процедура должна ожидать нажатия кнопки звонка, а во втором — нажатия одной из кнопок набора кода. Обе эти процедуры нужно объединить в одну. Объединенная процедура ожидания должна сначала опрашивать кнопку звонка, а затем проверять, не нажаты ли кнопки набора кода.

Если обнаружится нажатие кнопки звонка, выполняется алгоритм музыкальной шкатулки. Если нажатыми окажутся кнопки набора кода, то выполняется алгоритм электронного замка. Если ничего не нажато, цикл ожидания продолжается.

В том случае, если начнется выполнение музыкального алгоритма, программа будет целиком занята этим алгоритмом, пока кнопка звонка не будет отпущена. Если начнется выполнение алгоритма замка, то программа будет выполнять его до конца. То есть пока набранная комбинация не запишется в память или пока не закончится процедура проверки. И только тогда, когда выбранный алгоритм закончит свою работу, управление передается к объединенной процедуре ожидания.

Схема

За основу схемы объединенного устройства удобнее взять **схему электронного замка** (см. **рис. 1.17**). Она потребует минимальной доработки. Новая схема приведена на **рис. 1.14**. К старой схеме добавлены всего два элемента. **Во-первых,** это выходной каскад звука, подключенный к выходу РВ3. Именно этот выход используется в программе музыкальной шкатулки (**рис. 1.14**). **А во-вторых,** кнопка звонка, которую мы подключим к линии РВ6.

Обратите внимание, что звуковая часть и сам микроконтроллер питаются от напряжения +5 В. А электромагнит замка питается от отдельного источника +12 В. Напряжение питания, подаваемое на микроконтроллер, обязательно должно быть стабилизированным. Питание на электромагнит стабилизировать совершенно необязательно. Для повышения громкости звонка и для защиты от помех питание для звуковой схемы можно осуществлять от напряжения, поступающего на вход стабилизатора.

Обычно напряжение на входе стабилизатора равно +7...+11 В. Поэтому можно электромагнит и звуковую схему запитывать от одного нестабилизированного источника +12 В. Те же 12 В можно подавать на вход стабилизатора, с выхода которого снимать напряжение +5 В для микроконтроллера.

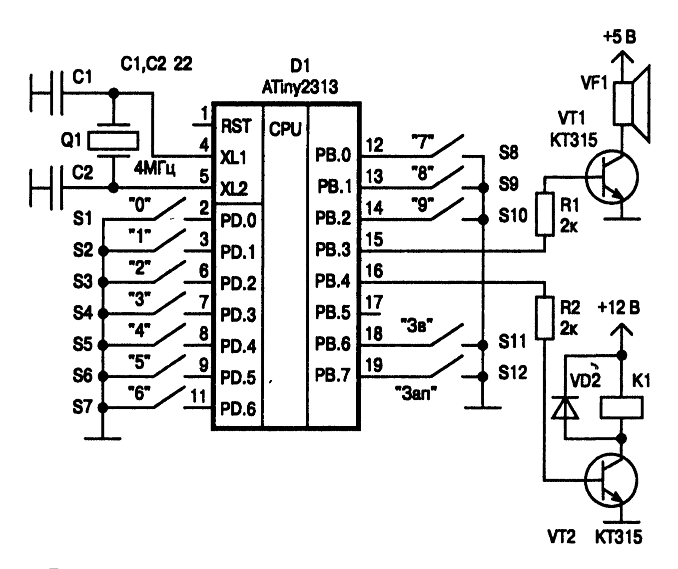


Рис. 1.19. *Схема усовершенствованного кодового замка*

Программа на Ассемблере

На **листинге 1.21** приведен возможный вариант объединения двух программ на Ассемблере. Новая программа, как и любая другая программа на Ассемблере, имеет свой модуль описания переменных и констант, свой модуль резервирования памяти, свою таблицу векторов прерывания и свою собственную основную часть.

Каждая из этих частей является результатом объединения аналогичных частей двух исходных программ. **Блок описаний переменных и констант** в новой программе занимает **строки 3—18**. При создании объединенного блока описаний учитывался тот факт, что в исходных программах широко используются одинаковые переменные. В объединенном блоке описаний каждая такая переменная описывается только один раз.

В **строках 19—22** находится **объединенный блок резервирования ОЗУ**. В **строке 21** резервируется буфер `bufr` для электронного замка. Такой же буфер резервировался и в исходной программе. В **строке 22** резервируется ячейка `melod`. Это новая ячейка, введенная в связи с доработкой алгоритма музыкальной шкатулки. В этой ячейке будет храниться текущий номер мелодии. При каждом последующем нажатии кнопки звонка этот номер будет изменяться.

В **строках 23—25** находится **блок резервирования ячеек в EEPROM**. Этот блок целиком взят из исходной программы электронного замка. В **строках 28—46** находится **таблица переопределения векторов прерываний**. Она тоже объединена из двух таблиц. Однако в связи с тем, что программа музыкальной шкатулки прерываний не использует, новая таблица полностью повторяет таблицу из программы электронного замка.

Модуль инициализации занимает **строки 47—60**. В данном случае в новый блок попали те команды, которые в исходных блоках обеих программ одинаковы. Это команды инициализации стека, инициализации портов ввода—вывода и инициализации (выключения) компаратора. Команды инициализации таймера для каждой из исходных программ отличаются друг от друга. Поэтому они исключены из общего блока инициализации. Инициализация таймера будет производиться поразному в зависимости от выбранного режима работы (звонок или замок).

В **строке 61** начинается **основная часть программы**. За основу этой части взята программа электронного замка. Программа звонка подключается к основной части путем доработки процедуры ожидания нажатия кнопки. В остальном программа замка не претерпела ни каких изменений.

Начинается основная программа с **инициализации таймера под задачи замка (строки 61—66)**. Затем в **строках 67, 68** записывается контрольное значение в буфер `codH+codL`. В качестве контрольного значения используется код состояния клавиатуры при полностью отпущенных кнопках. Этот код будет использоваться далее в цикле ожидания отпускания кнопок и в цикле ожидания нажатия кнопок. В **строках 69, 70** находится цикл ожидания отпускания кнопок. До сих пор программа замка не имела принципиальных изменений.

В **строках 71—75** находится **цикл ожидания нажатия кнопок**. Именно этот цикл изменен таким образом, что выполняет теперь комбинированный опрос не только кнопок ввода кода, но и кнопки звонка. И начинается цикл с опроса кнопки звонка (**строки 71—73**). В **строке 71** содержимое порта РВ считывается и помещается в регистр `temp`.

В **строке 72** проверяется значение бита номер шесть полученного числа. Именно этот бит отвечает за кнопку звонка. Если кнопка звонка нажата, то значение данного бита будет равно нулю. В этом случае команда условного перехода в **строке 73** передаст управление по метке `kk1`, где находится программа воспроизведения мелодий. Если кнопка звонка не нажата, то управление передается к **строке 74**. В **строках 74** и **75** находится знакомая нам процедура проверки кнопок набора кода.

Если ни одна из кнопок набора кода не нажата, то управление передается на начало комбинированного цикла (по метке `m1`), и цикл продолжается. При обнаружении факта нажатия одной или нескольких кнопок набора кода управление переходит к **строке 76.** С этой **строки** программа электронного замка не имеет отличий от оригинала.

Вся **оставшаяся часть программы** занимает **строки 76—127**. **Вспомогательные процедуры** занимают **строки 128— 185**.

Программа музыкального звонка расположена в **строках 186—295**. Ее пришлось немного доработать. Прежде всего, из программы был исключен модуль опроса клавиатуры, который в программе «Музыкальная шкатулка» (**листинг 1.17**) по нажатию одной из кнопок определял номер воспроизводимой мелодии. Вместо этой процедуры в программу введена другая, которая использует в качестве номера мелодии содержимое ячейки `melod`. При каждом нажатии кнопки номер мелодии увеличивается на единицу.

Если полученный таким образом номер превысит общее количество мелодий, то он сбрасывается в ноль, а подсчет мелодий начинается сначала. В новой шкатулке используется не семь, а восемь мелодий. Других принципиальных изменений музыкальная программа не имеет.

Начинается музыкальная программа с **модуля инициализации таймера (строки 186—189)**. В данном случае таймер настраивается на нужды программы воспроизведения звука. В **строках 190—195** расположен упомянутый выше модуль, определяющий номер мелодии. В **строке 190** текущее значение номера мелодии читается из ячейки памяти `melod` и помещается в регистр `count`.

В **строке 191** номер мелодии увеличивается на единицу. В **строке 192** полученный таким образом новый номер мелодии сравнивается с числом 8. Если номер меньше восьми, то оператор условного перехода в **строке 193** передает управление по метке `km2`, и **строка 194** не выполняется. Если номер мелодии равен или больше восьми, то выполняется **строка 194**, где номеру мелодии присваивается нулевое значение.

В **строке 195** новый номер мелодии записывается обратно в ячейку памяти melod. Там он хранится до следующего нажатия кнопки звонка. Кроме того, номер мелодии остается также и в регистре count. Именно в этот регистр помещала номер мелодии процедура сканирования клавиатуры в программе «музыкальная шкатулка».

После определения номера мелодии управление переходит к **строке 196**. В этой строке начинается уже известная нам **программа воспроизведения мелодий**. Она почти без изменений перенесена из программы (**листинг 1.17**). Но без изменений все же не обошлось.

Во-первых, пришлось переименовать некоторые метки, так как в программе электронного замка были использованы метки с теми же именами. Второе изменение связано с условием выхода из процедуры воспроизведения мелодии. В исходной программе для этого проверялось состояние сразу семи управляющих кнопок. И если хотя бы одна из них оставалась нажатой, воспроизведение мелодии продолжалось.

В нашем случае нам нужно проверить лишь одну кнопку — кнопку звонка. **Доработанная процедура проверки** расположена в **строках 204—206**. В **строке 204** читается содержимое порта PB и помещается в регистр temp. В **строке 205** проверяется разряд номер 6. Именно этот разряд связан с кнопкой звонка.

Если разряд равен нулю (кнопка нажата), то воспроизведение мелодии продолжается. Если разряд равен единице (кнопка отпущена), то оператор условного перехода в **строке 206** передает управление по метке km6, где происходит завершение процедуры формирования звука. После выключения звука команда безусловного перехода в **строке 223** передает управление по метке main, то есть на начало программы замка. И первыми операциями этой программы будет перенастройка таймера под требования этой основной задачи.

В **строках 289—295** находятся **таблицы всех мелодий**. В целях сокращения занимаемого места в данном примере все мелодии максимально сокращены. При повторении данной конструкции вы можете использовать мелодии целиком, взяв их из **листинга 1.17**. Еще проще скачать эту и все остальные программы в электронном виде с сайта http://book.microprocessor.by.ru. Там все восемь мелодий представлены в полном виде.

Листинг 1.21

```
;############################################
;##                                        ##
;##             Пример 11                  ##
;##           Кодовый замок                ##
;##       с музыкальным звонком            ##
;##                                        ##
;############################################

;------------------------ Псевдокоманды управления

.include "tn2313def.inc"  ; Присоединение файла описаний
.list                     ; Включение листинга

.def     drebL = R1       ; Буфер антидребезга младший байт
.def     drebH = R2       ; Буфер антидребезга старший байт
.def     temp1 = R3       ; Вспомогательный регистр
.def     temp = R16       ; Второй вспомогательный регистр
.def     data = R17       ; Регистр передачи данных
.def     flz = R18        ; Фаза работы замка
.def     count = R19      ; Регистр передачи данных
 def     addre = R20      ; Указатель адреса в EEPROM
.def     codL = R21       ; Временный буфер кода младший байт
.def     codH = R22       ; Временный буфер кода старший байт
.def     loop = R23       ; Регистр счетчика
.def     fnota = R24      ; Частота текущей ноты
.def     dnota = R25      ; Длительность текущей ноты

;------------------------ Определение констант

.equ     bsize = 60       ; Размер буфера для хранения кода
.equ     kzad  = 3000     ; Задержка при сканировании кнопок
.equ     kandr = 20       ; Константа антидребезга

;------------------------ Резервирование ячеек памяти (SRAM)

         .dseg            ; Выбираем сегмент ОЗУ
         .org     0x60    ; Устанавливаем текущий адрес сегмента

bufr:    .byte    bsize   ; Буфер для приема кода
melod:   .byte    1       ; Номер текущей мелодии

;------------------------ Резервирование ячеек памяти  (EEPROM)

         .eseg            ; Выбираем сегмент EEPROM
         .org     0x08    ; Устанавливаем текущий адрес сегмента

klen:    .byte    1       ; Ячейка для хранения длины кода
bufe:    .byte    bsize   ; Буфер для хранения кода

;------------------------ Начало программного кода

         .cseg            ; Выбор сегмента программного кода
         .org     0       ; Установка текущего адреса на ноль

start:   rjmp     init    ; Переход на начало программы
         reti             ; Внешнее прерывание 0
         reti             ; Внешнее прерывание 1
         reti             ; Таймер/счетчик 1, захват
         rjmp     propr   ; Таймер/счетчик 1, совпадение, канал А
         rjmp     propr   ; Таймер/счетчик 1, прерывание по переполнению
         reti             ; Таймер/счетчик 0, прерывание по переполнению
```

```
        reti            ; Прерывание UART прием завершен
        reti            ; Прерывание UART регистр данных пуст
        reti            ; Прерывание UART передача завершена
        reti            ; Прерывание по компаратору
        reti            ; Прерывание по изменению на любом контакте
        reti            ; Таймер/счетчик 1. Совпадение, канал B
        reti            ; Таймер/счетчик 0. Совпадение, канал B
        reti            ; Таймер/счетчик 0. Совпадение, канал A
        reti            ; USI готовность к старту
        reti            ; USI Переполнение
        reti            ; EEPROM Готовность
        reti            ; Переполнение охранного таймера

;*******************************************************
;*                                                     *
;*              Модуль инициализации                   *
;*                                                     *
;*******************************************************

init:
;------------------------- Инициализация стека

        ldi     temp, RAMEND    ; Выбор адреса вершины стека
        out     SPL, temp       ; Запись его в регистр стека

;------------------------- Инициализация портов В/В

        ldi     temp,0x18       ; Инициализация порта PB
        out     DDRB, temp
        ldi     temp,0xE7
        out     PORTB, temp

        ldi     temp, 0x7F      ; Инициализация порта PD
        out     PORTD, temp
        ldi     temp, 0
        out     DDRD, temp

;-------------------------- Инициализация (выключение) компаратора

        ldi     temp, 0x80
        out     ACSR, temp

;-------------------------- Номер мелодии

        ldi     temp,0          ; Сбрасываем в ноль
        sts     melod,temp


;*******************************************************
;*                                                     *
;*            Начало основной программы                *
;*                                                     *
;*******************************************************

main:
;-------------------------- Инициализация таймера

        ldi     temp, high(kzad )     ; Записываем коэффициент задержки
        out     OCR1AH, temp
        ldi     temp, low(kzad )
        out     OCR1AL, temp
        ldi     temp, 0x03            ; Выбор режима работы таймера
        out     TCCR1B, temp

        ldi     codL,0x7F       ; Код для сравнения (младший байт)
```

```
        ldi     codH,0x07       ; Код для сравнения (старший байт)

m0:     rcall   incod           ; Ввод и проверка кода клавиш
        brne    m0              ; Если хоть одна не нажата, продолжаем

m1:     in      temp,PINB       ; Проверка кнопки звонка
        sbrs    temp,6
        rjmp    kk1             ; Если нажата, переходим к звуковой части
        rcall   incod           ; Ввод и проверка кода клавиш
        breq    m1              ; Если не одна не нажата, продолжаем

m2:     ldi     ZH, high(bufr)   ; Установка указателя на начало буфера
        ldi     ZL, low(bufr)    ;
        clr     count            ; Сброс счетчика байт

;---------------------- Цикл ввода кода

m3:     cli                     ; Запрет всех прерываний
        ldi     data,1          ; Вызываем задержку первого типа
        rcall   wait            ; К подпрограмме задержки

m5:     rcall   incod           ; Ввод и проверка кода кнопок
        st      Z+,XL           ; Записываем его в буфер
        st      Z+,XH
        inc     count           ; Увеличение счетчика байтов
        inc     count
        cpi     count, bsize    ; Проверяем, не конец ли буфера
        brsh    m7              ; Если конец, завершаем ввод кода

        mov     codL, XL        ; Записываем код как старый
        mov     codH, XH

        ldi     data,2          ; Вызываем задержку третьего типа
        rcall   wait            ; К подпрограмме задержки

m6:     rcall   incod           ; Ввод и проверка кода кнопок
        brne    m3              ; Если изменилось, записываем в буфер
        cpi     flz,1           ; Проверка окончания фазы ввода кода
        brne    m6

m7:     sbic    PINB,7          ; Проверка состояния тумблера
        rjmp    m9              ; К процедуре проверки кода

; ----------------------------- Процедура записи кода

        mov     data, count     ; Помещаем длину кода в data
        ldi     addre, klen     ; Адрес в EEPROM для хранения длины кода
        rcall   eewr            ; Записываем в длину кода EEPROM

        ldi     addre, bufe      ; В регистр адреса начало буфера в EEPROM
        ldi     ZH, high(bufr)   ; В регистровую пару Z записываем
        ldi     ZL, low(bufr)    ; адрес начала буфера в ОЗУ

m8:     ld      data, Z+        ; Читаем очередной байт из ОЗУ
        rcall   eewr            ; Записываем в длину кода EEPROM
        dec     count           ; Декремент счетчика байтов
        brne    m8              ; Если не конец, продолжаем запись

        rjmp    m11             ; К процедуре открывания замка

; ----------------------------- Процедура проверки кода

m9:     ldi     addre, klen     ; Адрес хранения длины кода
```

```
        rcall   eerd            ; Чтение длины кода из EEPROM
        cp      count, data     ; Сравнение с новым значением
        brne    m13             ; Если не равны, к началу

        ldi     addre, bufe       ; В YL начало буфера в EEPROM
        ldi     ZH, high(bufr)    ; В регистровую пару Z записываем
        ldi     ZL, low(bufr)     ; адрес начала буфера в ОЗУ

m10:    rcall   eerd            ; Читаем очередной байт из EEPROM
        ld      temp, Z+        ; Читаем очередной байт из ОЗУ
        cp      data, temp      ; Сравниваем два байта разных кодов
        brne    m13             ; Если не равны, переходим к началу

        dec     count           ; Уменьшаем содержимое счетчика байтов
        brne    m10             ; Если не конец, продолжаем проверку

; ---------------------------- Процедура открывания замка

m11:    sbi     PORTB,4         ; Команда "Открыть замок"
        ldi     data,3          ; Вызываем задержку третьего типа
        rcall   wait
        cbi     PORTB,4         ; Команда "Закрыть замок"

m13:    rjmp    main            ; Перейти к началу


;*****************************************************
;*                                                   *
;*            Вспомогательные процедуры              *
;*                                                   *
;*****************************************************

; ----------------------------- Ввод и проверка 2 байтов с клавиатуры

incod:  push    count

        ldi     XL,0            ; Обнуление регистровой пары X
        ldi     XH,0

ic1:    ldi     count, kandr    ; Константа антидребезга
        mov     drebL, XL       ; Старое значение младший байт
        mov     drebH, XH       ; Старое значение старший байт

ic2:    in      XL,PIND         ; Вводим код (младший байт)
        cbr     XL,0x80         ; Накладываем маску
        in      XH,PINB         ; Вводим код (старший байт)
        cbr     XH,0xF8         ; Накладываем маску

ic3:    cp      XL, drebL       ; Сверяем младший байт
        brne    ic1             ; Если не равен, начинаем с начала
        cp      XH, drebH       ; Сверяем старший байт
        brne    ic1             ; Если не равен, начинаем с начала

ic4:    dec     count           ; Уменьшаем счетчик антидребезга
        brne    ic2             ; Если еще не конец, продолжаем

        cp      XL, codL        ; Сравнение с временным буфером
        brne    ic5             ; Если не равно, заканчиваем сравнение
        cp      XL, codH        ; Сравниваем старшие байты

ic5:    pop     count
        ret
```

```
; ------------------------------ Подпрограмма задержки

wait:   cpi     data,1          ; Проверяем код задержки
        brne    w1
        ldi     temp,0x40       ; Разрешаем прерывание по совпадению
        rjmp    w2
w1:     ldi     temp,0x80       ; Разрешаем прерывания по переполнению

w2:     out     TIMSK,temp      ; Записываем маску
        clr     temp
        out     TCNT1H,temp     ; Обнуляем таймер
        out     TCNT1L,temp

        ldi     flz,0           ; Сбрасываем флаг задержки
        sei                     ; Разрешаем прерывания
        cpi     data,2          ; Если это задержка 2-го типа
        breq    w4              ; Переходим к концу подпрограммы

w3:     cpi     flz,1           ; Ожидание окончания задержки
        brne    w3
        cli                     ; Запрещаем прерывания

w4:     ret                     ; Завершаем подпрограмму


; ------------------------------ Запись байта в ячейку EEPROM

eewr:   cli                     ; Запрещаем прерывания
        sbic    EECR, EEWE      ; Проверяем готовность EEPROM
        rjmp    eewr            ; Если не готов ждем
        out     EEAR, addre     ; Записываем адрес в регистр адреса
        out     EEDR, data      ; Записываем данные в регистр данных
        sbi     EECR, EEMWE     ; Устанавливаем бит разрешения записи
        sbi     EECR, EEWE      ; Устанавливаем бит записи
        inc     addre           ; Увеличиваем адрес в EEPROM
        ret                     ; Выход из подпрограммы


; ------------------------------ Чтение байта из ячейки EEPROM

eerd:   cli                     ; Запрещаем прерывания
        sbic    EECR, EEWE      ; Проверяем готовность EEPROM
        rjmp    eerd            ; Если не готов ждем
        out     EEAR, addre     ; Устанавливаем бит инициализации чтения
        sbi     EECR, EERE      ; Устанавливаем бит инициализации чтения
        in      data, EEDR      ; Помещаем прочитанный байт в data
        inc     addre           ; Увеличиваем адрес в EEPROM
        ret                     ; Выход из подпрограммы

;*****************************************************
;*                                                   *
;*         Процедура обработки прерываний            *
;*                                                   *
;*****************************************************

; -------------------------- Прерывание по совпадению

propr:  ldi     flz,1           ; Установка флага задержки
```

```
        reti                        ; Завершаем обработку прерывания

;#######################################################
;#                                                     #
;#                 Мелодичный звонок                   #
;#                                                     #
;#######################################################

kk1:
;-------------------------- Инициализация таймера T1

        ldi     temp, 0x09    ; Включаем режим СТС
        out     TCCR1B, temp
km1:    ldi     temp, 0x00    ; Выключаем звук
        out     TCCR1A, temp

;------------------------- Определение номера текущей мелодии

        lds     count,melod   ; Читаем код текущей мелодии
        inc     count         ; Увеличение номера мелодии на 1
        cpi     count,8       ; Проверка на последнюю мелодию
        brne    km2           ; Если не последняя, переход
        clr     count         ; Обнуление счетчика
km2:    sts     melod,count   ; Помещаем номер в ячейку памяти


;------------------------- Выбор мелодии

km3:    mov     YL, count          ; Вычисляем адрес, где
        ldi     ZL, low(tabm*2)    ; хранится начало мелодии
        ldi     ZH, high(tabm*2)
        rcall   addw               ; К подпрограмме 16-разрядного сложения

        lpm     XL, Z+             ; Извлекаем адреса из таблицы
        lpm     XH, Z              ; и помещаем в X

;-------------------------- Воспроизведение мелодии

km4:    mov     ZH, XH        ; Записываем в Z начало мелодии
        mov     ZL, XL

km5:    in      temp, PINB    ; Читаем содержимое порт B
        sbrc    temp, 6       ; Проверяем нажата ли еще кнопка звонка
        rjmp    km6           ; Если равно (кнопки отпущены) в начало

        lpm     temp, Z       ; Извлекаем код ноты
        opi     temp, 0xFF    ; Проверяем, не конец ли мелодии
        breq    m4            ; Если конец, начинаем мелодию сначала

        andi    temp, 0x1F    ; Выделяем код тона из кода ноты
        mov     fnota, temp   ; Записываем в регистр кода тона
        lpm     temp, Z+      ; Еще раз берем код ноты
        rol     temp          ; Производим четырехкратный сдвиг кода ноты
        rol     temp
        rol     temp
        rol     temp
        andi    temp, 0x07    ; Выделяем код длительности
        mov     dnota, temp   ; Помещаем ее в регистр длительности

        rcall   nota          ; К подпрограмме воспроизведения ноты

        rjmp    km5           ; В начало цикла (следующая нота)
```

```
km6:    ldi     temp, 0x00      ; Выключаем звук
        out     TCCR1A, temp
        rjmp    main            ; Переходим к началу


;*****************************************************
;*         Вспомогательные подпрограммы              *
;*****************************************************

; ---------------------- Подпрограмма 16-ти разрядного сложения
addw:   push    YH

        lsl     YL          ; Умножение первого слагаемого на 2
        ldi     YH, 0       ; Второй байт первого слагаемого = 0
        add     ZL, YL      ; Складываем два слагаемых
        adc     ZH, YH

        pop     YH
        ret

;---------------------- Подпрограмма исполнения одной ноты
nota:   push    ZH
        push    ZL
        push    YL
        push    temp

        cpi     fnota, 0x00         ; Проверка, не пауза ли
        breq    nt1                 ; Если пауза, переходим сразу к задержке

        mov     YL, fnota           ; Вычисляем адрес, где хранится
        ldi     ZL, low(tabkd*2)     ; коэффициент деления для текущей ноты
        ldi     ZH, high(tabkd*2)
        rcall   addw                ; К подпрограмме 16-разрядного сложения

        lpm     temp, Z+            ; Извлекаем мл. разряд КД для текущей ноты
        lpm     temp1, Z            ; Извлекаем ст. разряд КД для текущей ноты
        out     OCR1AH, temp1       ; Записать в старш. часть регистра совпадения
        out     OCR1AL, temp        ; Записать в младш. часть регистра совпадения

        ldi     temp, 0x40          ; Включить звук
        out     TCCR1A, temp

nt1:    rcall   wait1               ; К подпрограмме задержки

        ldi     temp, 0x00          ; Выключить звук
        out     TCCR1A, temp

        ldi     dnota,0             ; Сбрасываем задержку для паузы между нотами
        rcall   wait1               ; Пауза между нотами

        pop     temp                ; Завершение подпрограммы
        pop     YL
        pop     ZL
        pop     ZH
        ret

;---------------------- Подпрограмма формирования задержки
wait1:  push    ZH
        push    ZL
        push    YH
        push    YL
```

```
        mov     YL, dnota           ; Вычисляем адрес, где хранится
        ldi     ZL, low(tabz*2)     ; нужный коэффициент задержки
        ldi     ZH, high(tabz*2)
        rcall   addw                ; К подпрограмме 16-разрядного сложения

        lpm     YL, Z+              ; Читаем первый байт коэффициента задержки
        lpm     YH, Z               ; Читаем второй байт коэффициента задержки

        clr     ZL                  ; Обнуляем регистровую пару Z
        clr     ZH

        ; Цикл задержки
ww1:    ldi     loop,255            ; Пустой внутренний цикл
ww2:    dec     loop
        brne    ww2
        adiw    R30, 1              ; Увеличение регистровой пары Z на единицу
        cp      YL, ZL              ; Проверка младшего разряда
        brne    ww1
        cp      YH, ZH              ; Проверка старшего разряда
        brne    ww1

        pop     YL                  ; Завершение подпрограммы
        pop     YH
        pop     ZL
        pop     ZH
        ret


;****************************************************
;*          Таблица длительности задержек           *
;****************************************************

tabz:   .dw     128,256,512,1024,2048,4096, 8192

;****************************************************
;*          Таблица коэффициентов деления           *
;****************************************************

tabkd:  .dw     0
        .dw     4748,4480,4228,3992,3768,3556,3356,3168,2990,2822,2664, 2514
        .dw     2374,2240,2114,1996,1884,1778,1678,1584,1495,1411,1332, 1257
        .dw     1187,1120,1057, 998, 942, 889, 839, 792

;****************************************************
;*            Таблица начал всех мелодий            *
;****************************************************

tabm:   .dw     mel1*2,mel2*2,mel3*2,mel4*2
        .dw     mel5*2,mel6*2,mel7*2

;****************************************************
;*                  Таблица мелодий                 *
;****************************************************

;       В траве сидел кузнечик
mel1:   .db     109,104,109,104,109,108,108,96,108,104,108,104,108,109,109,96,255

;       Песенка крокодила Гены
mel2:   .db     109,110,141,102,104,105,102,109,110,141,104,105,107,104,109,110,255

;       В лесу родилась елочка
mel3:   .db     132,141,141,139,141,137,132,132,132,141,141,142,139,176,128,144,255
```

```
     ;      Happy births day to you
mel4:       .db      107,107,141,139,144,143,128,107,107,141,139,146,144,128,107,107,255

     ;      С чего начинается родина
mel5:       .db      99,175,109,107,106,102, 99,144,111,175,96,99,107,107,107,107,102,255

     ;      Песня из кинофильма "Веселые ребята"
mel6:       .db      105,109,112,149,116,64,80,148,114,64, 78,146,112,96,105,105,109,255

     ;      Улыбка
mel7:       .db      107,104,141,139,102,105,104,102,164,128,104,107,109,109,109,111,255
```

Программа на языке СИ

Объединение программ на языке СИ происходит точно таким же образом, как и на Ассемблер'е. То есть отдельно объединяются блоки описания, блоки инициализации и основные части программ. Возможный вариант комбинированной программы приведен в **листинге 1.22**. В **строках 3—13** описываются глобальные переменные, константы и массивы. Сюда вошли элементы, используемые в обеих объединяемых программах. В **строках 14—25** описаны массивы, используемые при генерации мелодий. Причем для экономии места все мелодии здесь также сокращены.

Строки 29—47 занимают вспомогательные функции для программы электронного замка. Сюда входят:

- две процедуры обработки прерываний (**строки 26—29**);
- процедура опроса клавиатуры (**строки 30—40**);
- процедура формирования задержки (**строки 41—47**).

Все эти процедуры перенесены из исходной программы без изменений.

Как и в программе на Ассемблере, за основу основной процедуры программы на СИ взята программа электронного замка. Поэтому музыкальная программа оформлена в виде отдельной функции `muz`, которая занимает **строки 48—67**. Функция представляет собой практически полную копию основной программы музыкальной шкатулки за исключением небольших изменений, о которых мы поговорим чуть дальше.

Все переменные, относящиеся именно **к задаче воспроизведения звука**, описываются внутри функции `muz` (**строки 49—51**). Лишь для хранения кода текущей мелодии используется глобальная переменная `melod` (описывается в **строке 9**).

Процедура сканирования кнопок из программы воспроизведения мелодии исключена и заменена небольшой процедуркой в **строке 67**. В этой строке происходит увеличение переменной `melod` на единицу и сравнение полученной величины с числом 8. Если `melod` больше либо равно 8, ему присваивается нулевое значение.

Второе изменение программы воспроизведения мелодии состоит в доработке процедуры проверки нажатой кнопки. Данная проверка занимает всего одну строку. В новой программе это **строка 54**. Теперь программа проверяет состояние шестого бита порта PB. Именно этот бит связан с кнопкой звонка. В остальном программа воспроизведения мелодий полностью повторяет аналогичную программу из **листинга 1.18**.

Программа кодового замка выполнена в виде главной процедуры программы и занимает **строки 68—105**. Текст этой программы почти полностью повторяет аналогичный текст в **листинге 1.20. Главное отличие** состоит в новой редакции процедуры ожидания отпускания кнопки. Она доработана и превращена в **комбинированную процедуру ожидания**.

В новой программе она занимает **строки 85—86**. Это все тот же цикл `while`, но на этот раз он не пустой. В данном случае цикл содержит один оператор. Это оператор `if` в **строке 86**. О том, что оператор `if` входит в тело цикла, свидетельствует отсутствие символа «точка с запятой» в конце **строки 85**. Для сравнения, в **строке 84** точка с запятой есть. Поэтому цикл в **строке 84** не имеет в своем теле никаких команд.

Внимание. ***Напоминаю, что в языке СИ конец строки не является признаком окончания команды. Операторы языка СИ отделяются друг от друга символом точки с запятой. Если точки с запятой нет, то все, что идет за очередным оператором, считается его продолжением.***

При этом вас не должно смущать даже наличие комментария в **строке 85**. Комментарий начинается символом «//» и заканчивается в конце строки. Для компилятора любой комментарий не существует. Поэтому команда в **строке 86** считается продолжением команды в **строке 85**. Как видите, простота и эффективность языка СИ имеет и свою обратную сторону. Наличие или отсутствие всего одного символа может кардинально изменить весь алгоритм программы.

Других изменений программа электронного замка не имеет. Нам даже не пришлось переименовывать метки.

Внимание. *В функции* `main` *и в функции* `muz` *новой программы используются одинаковые имена меток. В программе на Ассемблере в подобной ситуации нам пришлось переименовать метки одной из программ. В языке СИ метки, проставленные внутри одной из функций, не «видны» из другой. Поэтому переименовывать метки не обязательно.*

Однако в том случае, когда вы пишете программу с нуля, все же желательно не применять одинаковых имен в разных частях программы. Просто для того, чтобы самому не запутаться в этих именах.

Листинг 1.22

```
/*****************************************************
Project : Пример 11
Version : 1
Date    : 07.03.2006
Author  : Belov
Company : Home
Comments:
Кодовый замок с музыкальным звонком

Chip type           : ATtiny2313
Clock frequency     : 4,000000 MHz
Memory model        : Tiny
Data Stack size     : 32
*****************************************************/

#include <tiny2313.h>
#include <delay.h>

#define klfree 0x77F                  // Код состояния при полностью отпущенных кнопках
#define kzad 3000                     // Код задержки при сканировании
#define kandr 20                      // Константа антидребезга
#define bsize 30                      // Размер буфера для хранения кода

unsigned char flz;                    // Флаг задержки
unsigned int bufr[bsize];             // Буфер в ОЗУ для хранения кода
unsigned char melod;                  // Текущий номер мелодии

#pragma warn-                         // Отмена предупреждающих сообщений
eeprom unsigned char klen;            // Ячейка для хранения длины кода
eeprom unsigned int bufe[bsize];      // Буфер в EEPROM для хранения кода
#pragma warn-                         // Разрешение предупреждающих сообщений
```

```
// Объявление и инициализация массивов

// Таблица задержек
flash unsigned int tabz[] =  {16,32,64,128,256,512,1024};

// Массив коэффициентов деления
flash unsigned int tabkd[] = {0,4748,4480,4228,3992,3768,3556,3356,3168,2990,2822,
            2664,2514,2374,2240,2114,1996,1884,1778,1678,1584,1495,1411,1332,1257,
            1187,1120,1057, 998,942,889,839,792};

// Таблицы мелодий
flash unsigned char mel1[] = {109,104,109,104,109,108,108,96,108,104,108,104,255};
flash unsigned char mel2[] = {109,110,141,102,104,105,102,109,110,141,104,105, 255};
flash unsigned char mel3[] = {132,141,141,139,141,137,132,132,132,141,141,142,255};
flash unsigned char mel4[] = {107,107,141,139,144,143,128,107,107,141,139,146,255};
flash unsigned char mel5[] = {99,175,109,107,106,102,99,144,111,175,96,99,107, 255};
flash unsigned char mel6[] = {105,109,112,149,116,64,80,148,114,64,78,146,112,255};
flash unsigned char mel7[] = {107,104,141,139,102,105,104,102,164,128,104,107,255};

// Таблица начал всех мелодий
flash unsigned char *tabm[] = {mel1, mel2, mel3, mel4, mel5, mel6, mel7};


// Прерывание по переполнению Таймера 1
interrupt [TIM1_OVF] void timer1_ovf_isr(void)
{
    flz=1;
}

// Прерывание по совпадению в канале А Таймера 1
interrupt [TIM1_COMPA] void timer1_compa_isr(void)
{
   flz=1;
}

// Функция опроса клавиатуры и антидребезга
unsigned int incod (void)
{
unsigned int cod0=0;                    // Создаем локальные переменные
unsigned int cod1;
unsigned char k;

for (k=0; k<kandr; k++)                 // Цикл антидребезга
{
    cod1=PINB&0x7;                      // Формируем первый байт кода
    cod1=(cod1<<8)+(PIND&0x7F);         // Формируем полный код состояния клавиатуры
    if (cod0!=cod1)                     // Сравниваем с первоначальным кодом
    {
        k=0;                            // Если не равны, сбрасываем счетчик
        cod0=cod1;                      // Новое значение первоначального кода
    }
}
   return cod1;
}

// Процедура формирования задержки
void wait (unsigned char kodz)
{
if (kodz==1) TIMSK=0x40;                // Выбор маски прерываний по таймеру
else TIMSK=0x80;
TCNT1=0;                                // Обнуление таймера
```

```
flz=0;                              // Сброс флага задержки
#asm("sei");                        // Разрешаем прерывания
if (kodz!=2) while(flz==0);         // Цикл задержки
}

// Музыкальная программа
void muz (void)
{
unsigned char fnota;                // Код тона ноты
unsigned char dnota;                // Код длительности ноты
flash unsigned char *nota;          // Ссылка на текущую ноту

            TCCR1A=0x00;            // Выключение звука

            // Воспроизведение мелодии
m3:         nota = tabm[melod];     // Устанавливаем указатель на первую ноту
m4:         if (PINB.6!=0) goto m2; // Если кнопка звонка не нажата, закончить
            if (*nota==0xFF) goto m3; // Проверка на конец мелодии
            fnota = (*nota)&0x1F;   // Определяем код тона
            dnota = ((*nota)>>5)&0x07; // Определяем код длительности
            if (fnota==0) goto m5;  // Если пауза не воспроизводим звук
            OCR1A=tabkd[fnota];     // Программируем частоту звука
            TCCR1A=0x40;            // Включаем звук
m5:         delay_ms (tabz[dnota]); // Формируем задержку
            TCCR1A=0x00;            // Выключаем звук
            delay_ms (tabz[0]);     // Задержка между нотами
            nota++;                 // Перемещаем указатель на следующую ноту
            goto m4;                // К началу цикла
m2:         TCCR1A=0x00;            // Выключаем звук
            if (++melod>=8) melod=0; // Увеличиваем счетчик мелодий
}


// Основная функция
void main(void)
{
unsigned char ii;         // Указатель массива
unsigned char i;          // Вспомогательный указатель
unsigned int codS;        // Старый код
PORTB=0xE7;               // Порт B
DDRB=0x18;
PORTD=0x7F;               // Порт D
DDRD=0x00;
TCCR1A=0x00;              // Таймер/Счетчик 1
TCCR1B=0x03;
TCNT1=0;
OCR1A=kzad;
ACSR=0x80;                // Аналоговый компаратор
melod=0;                  // Сбрасываем счетчик мелодий

while (1)
{
m1:         while (incod() != klfree);      // Ожидание отпускания кнопок
            while (incod() == klfree)       // Ожидание нажатия кнопок
              if (PINB.6==0) muz();         // К музыкальному звонку
            TCCR1B=0x03;                    // Настройка таймера
            ii=0;                           // Сброс счетчика байтов
m2:         #asm("cli");                    // Запрещаем прерывания
            wait(1);                        // Задержка 1-го типа
            codS=incod();                   // Ввод кода и запись, как старого
            bufr[ii++]=codS;                // Запись очередного кода в буфер
```

8 Зак 911

```
            if (ii>=bsize) goto m4;            // Проверка конца буфера

            wait(2);                           // Задержка 2-го типа
m3:         if (incod() != codS) goto m2;      // Проверка, не изменилось ли состояние
            if (flz==0) goto m3;               // Проверка флага окончания задержки

m4:         if (PINB.7==1) goto comp;          // Проверка переключателя режимов

//------------------------------ Запись кода в EEPROM

            klen=ii;                                   // Запись длины кода
            for (i=0; i<ii; i++) bufe[i]=bufr[i];      // Запись всех байтов кода
            goto zamok;                                // К процедуре открывания замка

//------------------------------ Проверка кода

comp:       if (klen!=ii) goto m1;            // Проверка длины кода
            for (i=0; i<ii; i++)  if (bufe[i]!=bufr[i]) goto m1;    // Проверка кода

//------------------------------ Открывание замка

zamok:      PORTB.4=1;            // Открываем замок
            wait(3);              // Задержка 3-го типа
            PORTB.4=0;            // Закрываем замок
}
}
```

ГЛАВА 2

Отладка и трансляция программ

Глава дает представление о технологии ввода в компьютер текста программы, редактирования и предварительной проверки программ, вводит в понятия отладки и отладчика.

Подробно описывается процесс транслирования программы в машинные коды и записи этих кодов в программную память микросхемы микроконтроллера.

2.1. Программная среда AVR Studio

2.1.1. Общие сведения

Отладка программы

В предыдущей главе мы научились создавать программы для микроконтроллеров. Однако, как уже говорилось ранее, для того, чтобы написанная программа превратилась в результирующий код и заработала в конкретном микропроцессорном устройстве, ее нужно **оттранслировать и «зашить» в программную память микроконтроллера**.

Однако существует еще один **важный аспект** этой задачи. Дело в том, что при написании реальной программы, особенно если программа реализует достаточно сложный алгоритм, невозможно избежать ошибок. Ошибки могут быть самые разные. От простой синтаксической ошибки в написании какой-либо команды до хитрых структурных ошибок, которые иногда очень трудно обнаружить.

В любом случае при написании программ обычно нельзя обойтись без **процедуры отладки**. Отладка выполняется на компьютере при помощи специальной инструментальной программы — **отладчика**. Отладчик позволяет пошагово выполнять отлаживаемую программу, а также выполняет ее поэтапно с использованием так называемых точек останова.

В процессе выполнения программы под управлением отладчика программист может на экране компьютера:

- видеть содержимое любого регистра микроконтроллера;
- видеть содержимое ОЗУ и EEPROM;
- наблюдать за последовательностью выполнения команд, контролируя правильность отработки условных и безусловных переходов;
- наблюдать за работой таймеров, отработкой прерываний.

В процессе отладки программист также может наблюдать логические уровни на любом внешнем выходе микроконтроллера. А также имитировать изменение сигналов на любом входе. Процесс отладки позволяет программисту убедиться в том, что разрабатываемая им программа работает так, как он задумал. Большинство ошибок в программе обнаруживаются именно в процессе отладки.

Существует **три основных вида отладчиков:**

- программные;
- аппаратные;
- комбинированные программно-аппаратные.

Программный отладчик

Определение. ***Программный отладчик** — это компьютерная программа, которая имитирует работу процессора на экране компьютера. Она не требует наличие реальной микросхемы или дополнительных внешних устройств и позволяет отладить программу чисто виртуально.*

Однако программный отладчик позволяет проверить только логику работы программы. При помощи такого отладчика невозможно проверить работу схемы в режиме реального времени или работу всего микропроцессорного устройства в комплексе. То есть невозможно гарантировать правильную работу и всех подключенных к микроконтроллеру дополнительных микросхем и элементов.

Аппаратный отладчик

Определение. *Второй вид отладчиков — **аппаратный отладчик**. Основа такого отладчика — специальная плата, подключаемая к компьютеру, работающая под его управлением и имитирующая работу реальной микросхемы микроконтроллера. Плата имеет выводы, соответствующие выводам реальной микросхемы, на которых в процессе отладки появляются реальные сигналы.*

При помощи этих выводов отладочная плата может быть включена в реальную схему. Возникающие в процессе отладки электрические сигналы можно наблюдать при помощи осциллографа. Можно нажимать реальные кнопки и наблюдать работу светодиодов и других индикаторов.

В то же самое время на экране компьютера мы так же, как и в предыдущем случае, можем видеть всю информации об отлаживаемой программе:

- наблюдать содержимое регистров, ОЗУ, портов ввода-вывода;
- контролировать ход выполнения программы.

В аппаратном отладчике мы можем так же, как и в программном, выполнять программу в пошаговом режиме и применять точки останова. **Недостатком** аппаратного отладчика является его высокая стоимость.

Полнофункциональные программные имитаторы электронных устройств

Существует и **третий вид отладчиков**. Это полнофункциональные программные имитаторы электронных устройств. Такие программы позволяют на экране компьютера «собрать» любую электронную схему, включающую в себя самые разные электронные компоненты:

- транзисторы;
- резисторы;
- конденсаторы;
- операционные усилители;
- логические и цифровые микросхемы, в том числе и микроконтроллеры.

Такие программы обычно содержат обширные базы электронных компонентов и конструктор электронных схем. Собрав схему, вы можете виртуально записать в память микроконтроллера вашу программу, а затем «запустить» всю схему в работу.

Для контроля результатов работы схемы имитатор имеет **виртуальные вольтметры, амперметры** и **осциллографы**, которые вы можете «подключать» к любой точке вашей схемы, «измерять» различные напряжения, а также «снимать» временные диаграммы.

Такие программы в настоящее время получают все большее распространение. Они позволяют разработать любую схему с микроконтроллером или без него, без использования паяльника и реальных деталей. На экране компьютера можно полностью отладить свою схему и лишь потом браться за паяльник.

Недостатком данного отладчика является то, что он требует значительных вычислительных ресурсов. Особенно в том случае, когда отлаживается схема, включающая как микроконтроллер, так и некоторую аналоговую часть. Кроме того, имитатор не всегда верно имитирует работу некоторых устройств. Однако подобные программы имеют очень большие перспективы. В рамках данной

книги я не буду рассматривать подобную программу, так как такая задача достойна отдельной книги.

Внутренний отладчик микроконтроллеров AVR

Еще один аппаратный способ отладки заложен конструктивно в некоторые модели микроконтроллеров AVR. В частности, микроконтроллер ATiny2313 поддерживает такой способ отладки (см. [5]).

Для обеспечения возможности аппаратной отладки такие микроконтроллеры имеют, во-первых, **специальную однопроводную линию debugWIRE**, которая обычно совмещена с входом RESET. Эта линия используется специальной платой — отладчиком для управления микроконтроллером в процессе отладки. Кроме того, в систему команд такого микроконтроллера включена команда `break`, которая может использоваться **для создания программных точек останова**.

Для того, чтобы использовать подобный режим отладки, необходимо иметь в своем распоряжении **специальную отладочную плату**, которая должна поддерживать этот режим. Кроме того, подобный режим должна поддерживать и **инструментальная программа-отладчик**.

В процессе отладки программист проставляет на экране компьютера в нужных местах отлаживаемой программы точки останова. Затем он запускает эту программу под управлением отладчика. Отладчик автоматически вставляет в отлаживаемую программу команды `break` в тех местах, где программист поставил точки останова. А команды, которые должны быть записаны в месте вставки команд `break`, запоминает в своей памяти.

Затем он автоматически «прошивает» полученный таким образом текст программы в программную память отлаживаемого микроконтроллера и запускает ее в работу. Микроконтроллер выполняет заложенную в него программу до тех пор, пока не встретится команда `break`. Получив эту команду, микроконтроллер приостанавливает выполнение программы и передает управление отладчику.

Далее отладчик управляет микроконтроллером при помощи **интерфейса debugWIRE**. Этот интерфейс позволяет считать содержимое всех регистров микроконтроллера и других видов памяти. Прочитанная информация отображается на экране компьютера. Затем отладчик ждет команд от оператора. Под управлением отладчика микроконтроллер может принудительно выполнить любую команду из своей системы команд.

Это дает возможность легко реализовать пошаговое выполнение программы, а также выполнение тех команд, которые были заменены на `break`. Все управление осуществляется посредством интерфейса debugWIRE, который позволяет передавать информацию как от отладчика в микроконтроллер, так и в обратном направлении.

Преимуществом такого способа отладки является то, что в данном случае происходит не имитация микроконтроллера, а используется реальная микросхема. При этом работа в режиме отладки наиболее полно приближается к реальному режиму работы.

Недостаток — частое «перешивание» программной памяти микроконтроллера. Изменять содержимое этой памяти приходится каждый раз при установке новых или снятии старых точек останова. Если учесть, что допустимое количество перезаписи программной памяти составляет 10000 циклов, то при длительном процессе отладки это количество может исчерпаться, и микросхема выйдет из строя.

Программная среда «AVR Studio»

Фирма *Atmel*, разработчик микроконтроллеров AVR, очень хорошо позаботилась о сопровождении своей продукции. Для написания программ, их отладки, трансляции и прошивки в память микроконтроллера фирма разработала и бесплатно распространяет **специализированную среду разработчика под названием «AVR Studio»**. Инсталляционный пакет этой инструментальной программы можно свободно скачать с сайта фирмы. Адрес страницы для скачивания программ:

http://www.atmel.ru/Software/Software.htm.

Программная среда «AVR Studio» — это мощный современный программный продукт, позволяющий производить все этапы разработки программ для любых микроконтроллеров серии AVR. Пакет включает в себя специализированный текстовый редактор для написания программ, мощный программный отладчик.

Кроме того, «AVR Studio» позволяет управлять целым рядом подключаемых к компьютеру внешних устройств, позволяющих выполнять аппаратную отладку, а также программирование («прошивку») микросхем AVR.

Познакомимся подробнее с этим удобным программным инструментом для программистов. Программная среда «AVR Studio» работает не просто с программами, а с проектами. Для каждого проекта должен быть отведен свой отдельный каталог на жестком

диске. В AVR Studio одновременно может быть загружен только один проект.

При загрузке нового проекта предыдущий проект автоматически выгружается. Проект содержит всю информацию о разрабатываемой программе и применяемом микроконтроллере. Он состоит из целого набора файлов.

Главный из них — **файл проекта**. Он имеет расширение aps. Файл проекта содержит сведения о типе процессора, частоте тактового генератора и т. д. Он также содержит описание всех остальных файлов, входящих в проект. Все эти сведения используются при отладке и трансляции программы.

Кроме файла aps, проект должен содержать хотя бы один **файл с текстом программы.** Такой файл имеет расширение asm. Недостаточно просто поместить файл asm в директорию проекта. Его нужно еще включить в проект. Как это делается, мы увидим чуть позже. Проект может содержать несколько файлов asm. При этом один из них является главным. Остальные могут вызываться из главного при помощи оператора .include. На этом заканчивается список файлов проекта, которые создаются при участии программиста.

Но типичный проект имеет гораздо больше файлов. Остальные файлы проекта появляются в процессе трансляции. Если ваша программа не содержит критических ошибок и процесс трансляции прошел успешно, то в директории проекта автоматически появляются следующие файлы: файл, содержащий результирующий код трансляции в hex формате, файл map, содержащий все символьные имена транслируемой программы со своими значениями, листинг трансляции (lst) и другие вспомогательные файлы. Однако для нас будет важен лишь hex-файл (файл с расширением hex). Именно он будет служить источником данных при прошивке программы в программную память микроконтроллера.

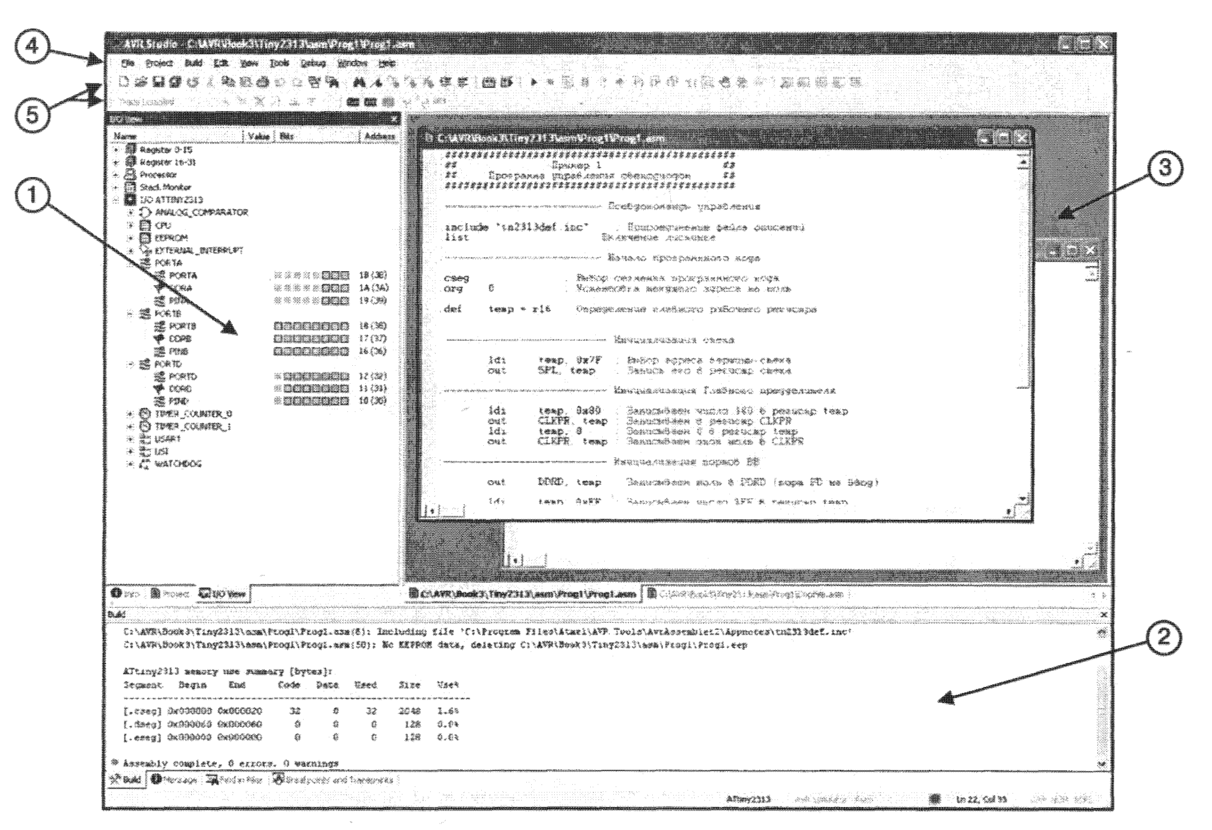


Рис. 2.1. *Программа AVR Studio*

2.1.2. Описание интерфейса

Главная панель программы «AVR Studio»

На **рис. 2.1** показано, как выглядит главная панель программы «AVR Studio». На самом деле «AVR Studio» имеет очень гибкий интерфейс, и внешний вид может сильно отличаться от варианта, показанного на рисунке. Но мы будем рассматривать случай, когда выбраны установки по умолчанию.

Главная панель программы AVR Studio разделена на три основных окна. На **рис. 2.1** они обозначены цифрами 1, 2 и 3. Первые два окна — вспомогательные. Окно 1 предоставляет нам полную информацию о текущем проекте. По умолчанию это окно включает в себя три вкладки. «Корешки» этих вкладок вы можете видеть в нижней части окна.

Первая вкладка называется «Info». Она содержит справочную информацию по используемому микроконтроллеру, такую как описание векторов прерываний, описание выводов для разных корпусов и краткое описание регистров.

Следующая вкладка называется «Project». Она содержит полную информацию по текущему загруженному проекту. Информация представлена в виде дерева. Разные ветви этого дерева описывают все исходные и результирующие файлы проекта, все метки, процедуры и присоединяемые файлы.

Последняя вкладка окна номер 1 называется «I/O View» (просмотр ввода—вывода). Это самая полезная вкладка. На ней в графическом виде показаны все ресурсы микроконтроллера:

- порты ввода—вывода;
- таймеры;
- компараторы;
- АЦП;
- регистры общего назначения и т. д.

Вся информация также представлена **в виде дерева**. Каждая «ветвь» этого дерева — это отдельный элемент. Если какой-либо элемент состоит из других элементов, то его можно раскрыть и увидеть эти элементы.

Элементы, появляющиеся в результате раскрытия ветви, в свою очередь также могут быть раскрыты, если они имеют свое содержимое. На **рис. 2.2** в увеличенном виде показано дерево ресурсов

микроконтроллера ATiny2313. На рисунке несколько ветвей специально раскрыты, чтобы можно было увидеть их состав.

Если какая-либо ветвь может быть раскрыта, то в своем основании она имеет квадратик с плюсиком внутри. Двойной щелчок на этом плюсике раскрывает ветвь. В раскрытой ветви плюсик превращается в минус. Повторный двойной щелчок по квадратику закрывает раскрытую ветвь.

На **рис. 2.2** для наглядности раскрыты ветви всех трех портов ввода—вывода и регистры, связанные с EEPROM. Вы можете видеть:

- полный состав управляющих регистров для каждого из устройств;
- их названия и адреса;
- состав и название каждого бита (если биты имеют свои названия).

Для наглядности на **рис. 2.2** раскрыта ветвь, соответствующая регистру EECR, и вы можете видеть все его биты.

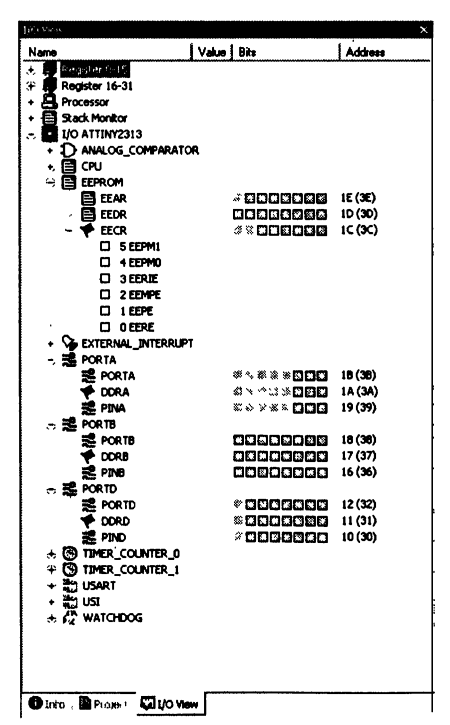


Рис. 2.2. *Окно ресурсов микроконтроллера*

В процессе отладки в этом окне вы увидите не только название и состав всех ресурсов, но и их содержимое. **Содержимое** будет отображаться как в шестнадцатиричном виде, так и путем затемнения квадратиков, отображающих отдельные биты конкретных регистров.

Затемненный квадратик означает, что бит равен единице. **Светлый квадратик** говорит о том, что бит равен нулю. Вы также можете оперативно менять это содержимое прямо в этом окне. Для изменения значения бита достаточно двойного щелчка мышки в соответствующем квадратике. Существуют и другие способы изменения содержимого различных регистров и ячеек памяти в процессе отладки.

В нижней части главной панели находится **второе вспомогательное окно** (окно 2 на **рис. 2.1**). Это окно служит. в основном, для вывода различных сообщений. Оно также содержит ряд вкладок. По умолчанию их четыре. **Первая вкладка называется «Build»**. На вкладке «Build» отражается процесс трансляции. На эту вкладку выводятся сообщения об различных этапах трансляции, сообщения о синтаксических ошибках и различные предупреждения (Warnings).

Если трансляция заканчивается нормально (отсутствуют критические ошибки), то сюда же выводятся статистические данные о полученном результирующем коде. Эти данные показывают размеры и процент использования всех видов памяти микроконтроллера. **Например**, после трансляции нашего примера №11 (**листинг 1.21**) программа выдаст следующее сообщение:

```
ATtiny2313 memory use summary [bytes]:
Segment Begin End Code Data Used Size Use%
---------------------------------------------
[.cseg] 0x000000 0x0004f2 508 758 1266 2048 61.8%
[.dseg] 0x000060 0x00009d   0  61   61  128 47.7%
[.eseg] 0x000008 0x000045   0  61   61  128 47.7%

Assembly complete, 0 errors. 0 warnings
```

Сообщение означает, что в программном сегменте использованы ячейки с адреса 0x000000 по адрес 0x0004f2. При этом собственно код программы занимает 508 байт. Данные в программной памяти занимают 758 байт. Всего использовано в программной памяти 1266 байт (сумма предыдущих двух чисел). Размер программной памяти для этого микроконтроллера составляет 2048 байт. Процент использования программной памяти 61,8%.

Точно такие же сведения приведены для памяти данных (ОЗУ) и для EEPROM. Естественно, что два последних вида памяти не содержат программного кода. Поэтому в соответствующем столбике стоят нули. Последняя строка содержит сообщения об ошибках. В данном случае сообщение переводится так: «Ассемблирование прошло успешно, 0 ошибок, 0 предупреждений».

Следующая вкладка второго окна называется «Message». Здесь выводятся разные системные сообщения о загрузке модулей программы и т. п.

Третья вкладка второго окна называется «Find in Files» (поиск в файлах). В этом окне отражаются результаты выполнения команды «Поиск в Файлах». Эта команда позволяет производить поиск заданной последовательности символов сразу во всех файлах проекта. По окончании поиска во вкладке «Find in Files» отражаются все найденные вхождения с указанием имени файла и строки, где найдена искомая последовательность.

Последняя вкладка называется «Breakpoints and Tracepoints» (Точки останова и точки трассировки). Эти точки проставляются в тексте программы перед началом процесса отладки и дублируются в данном окне. Как проставлять точки останова, мы узнаем чуть позже.

Точки останова используются для того, чтобы приостановить выполнение программы в том или ином месте программы для того, чтобы убедиться, что программа выполняется правильно. При создании точки останова в тексте программы она автоматически появляется во вкладке «Breakpoints and Tracepoints».

Вкладка позволяет увидеть все точки останова программы в одном месте. Кроме того, на вкладке против каждой записи, описывающей точку останова, автоматически появляется «Check box» (поле выбора), при помощи которого можно в любой момент временно отключить любую точку останова.

Точки трассировки используются для управления процессом трассировки.

Определение. *Трассировка — это особый вид отладочного процесса, когда программа запускается и выполняется в автоматическом режиме.*

Но в процессе работы она оставляет сообщения в специальном окне. Сообщения отражают каждый шаг выполняемой программы. Точки трассировки могут отменить и заново разрешить трассировку на разных участках программы.

Программная среда «AVR Studio» поддерживает трассировку только при работе с отладочной платой ICE50. Это достаточно дорогое устройство. Поэтому в этой книге мы остановимся лишь на программном отладчике без применения каких-либо аппаратных средств отладки.

Поверьте, этого вполне достаточно для разработки микропроцессорных устройств практически любой сложности. Аппаратные отладчики необходимы в условиях промышленного производства для ускорения работ по разработке новых изделий.

Любую из вкладок любого вышеописанного окна можно скрыть или, наоборот, превратить в отдельное свободно перемещаемое окно. Для этого достаточно щелкнуть правой клавишей мыши по заголовку соответствующей вкладки и выбрать в открывшемся меню нужный режим. Пункт «Hide» этого меню означает «Скрытое» (невидимое), «Floating» означает «Свободное» (Перемещающееся), «Docking» — «Закрепленное».

Для некоторых пользователей бывает затруднительно вернуть вкладку на место после того, как она превратится в свободно перемещаемое окно. В программе «AVR Studio» используется нестандартный довольно оригинальный механизм управления окнами. Предположим, что мы случайно превратили в плавающее окно вкладку «Breakpoints and Tracepoints» окна номер два. Посмотрим, как можно поставить ее на место.

Если перемещать это окно при помощи мыши (удерживая его за заголовок), то на основной панели программы появляются специальные указатели размещения, так как это показано на **рис. 2.3**. Они представляют собой стилизованные стрелки синего цвета, расположенные по всему полю главного окна программы. Одновременно появляются восемь таких стрелок. Четыре из них объединены в центральный блок, в который включена еще и круглая кнопка посередине.

Этот блок автоматически располагается в центре того окна, в пределах которого в данный момент перемещается курсор. На **рис. 2.3** этот указатель расположен в центре окна номер два. Оставшиеся четыре стрелки располагаются сверху, снизу, справа и слева основного окна программы. Достаточно переместить курсор мыши

вместе с перемещаемым плавающим окном на одну из этих стрелок и отпустить кнопку мыши, и окно тут же встроится в одно из вспомогательных окон программы в виде вкладки либо образует новое вспомогательное окно. Причем вы еще до отпускания кнопки мыши можете увидеть, куда попадет окно.

Как только ваш курсор совместится с одной из стрелок, программа закрасит синим цветом эту область. **Поэкспериментируйте** сами с размещением окон. Помните только, что стрелки превращают ваше плавающее окно в еще одно фиксированное дополнительное окно в разных частях интерфейса. А круглая кнопка посреди центрального блока превращает плавающее окно в дополнительную вкладку уже существующего окна. Именно при помощи этой кнопки оторванное от своего привычного места плавающее окно на **рис. 2.3** можно вернуть на свое место.

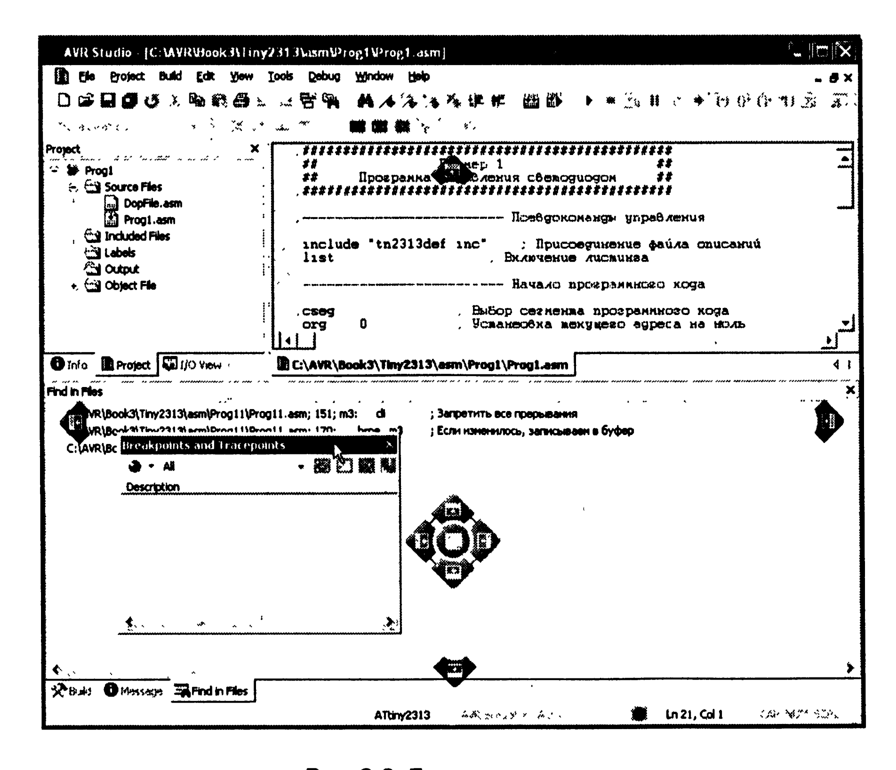


Рис. 2.3. *Перемещение окна*

Дополнительные окна 1 и 2 позволяют легко изменять свои размеры. Для изменения размера достаточно перетащить границу окна при помощи мыши. Можно даже скрыть любое из этих окон, закрыв все его вкладки. Закрыть вкладку можно двумя способами. Либо щелкнуть по ее «корешку» правой кнопкой мыши, а в появившемся меню выбрать пункт «Hide». Либо щелкнуть мышью в крестик в верхнем правом углу вкладки. Открыть закрытые вкладки можно при помощи меню «View/Toolbars».

Особую роль играет **окно 3**. Это даже не окно, а оставшаяся часть от главного окна программы. Если закрыть окна 1 и 2, окно 3 займет все пространство программной панели. В окне 3 появляются различные рабочие окна.

Во-первых, это окна с текстами программ на Ассемблере. А **во-вторых**, здесь могут появляться окна любых открытых программой файлов. Это могут быть текстовые файлы или файлы других программ. Каждый такой файл по умолчанию открывается в виде отдельного плавающего окна. Для определенности будем называть такие окна текстовыми окнами. Текстовые окна будут «плавать» только внутри окна 3.

Для каждого нового текстового окна в нижней части окна 3 появляется «корешок», при помощи которого можно быстро перейти к нужному окну, если оно не находится на переднем плане. Если произвести двойной щелчок левой кнопкой мыши по заголовку любого текстового окна, оно раскроется на всю ширину окна 3. Иногда именно так удобно работать с тестами программ.

В окне 3 можно открывать не только все тексты ассемблерных программ текущего проекта, но и тексты программ других проектов, а также тексты программ, написанных на других языках программирования. Такой прием очень удобен, если нужно переделать программу, написанную для старого микроконтроллера на старой версии Ассемблера на новый лад. Все открытые текстовые окна запоминаются и затем открываются автоматически при открытии проекта.

Любое текстовое окно имеет **подсветку синтаксиса**. Разные части помещенного туда текста программы подсвечиваются разными цветами. Так, все операторы Ассемблера высвечиваются **голубым** цветом. Комментарии выделяются **зеленым**. Остальной текст (параметры команд, псевдооператоры, метки, переменные и константы) остается **черным**. Это очень удобно. Если написанный вами оператор окрасился в голубой цвет, то это значит, что вы не ошиблись в синтаксисе. Если вы написали комментарий, но перед текстом комментария забыли поставить точку с запятой, то этот коммента-

рий не окрасится в зеленый цвет. Таким образом, многие ошибки видны уже в процессе написания программы.

Кроме двух вспомогательных и одного основного окна, главная панель программы имеет **строку меню** (отмечена цифрой **4** на **рис. 2.1**), а также несколько инструментальных панелей (отмечены цифрой **5**). Как и в любой другой программе под Windows, при помощи меню вызываются все функции программы AVR Studio и переключаются все ее режимы. Панели инструментов дублируют часто используемые функции меню.

2.1.3. Создание проекта

Предположим, что программа AVR Studio установлена на ваш компьютер, запущена и находится в исходном состоянии (все вкладки окон 1 и 2 пусты, окно 3 не содержит открытых файлов). Приступим к созданию нового проекта.

Для этого выберем в меню «Project» пункт «New Project». На экране появится окно построителя. В поле «Project Type:» выбираем **тип будущего проекта**. Программа предлагает два варианта:

- проект на Ассемблере (Atmel AVR Assembler);
- проект на языке СИ++ (AVR GCC).

Выбираем **Ассемблер**. Затем в поле «Project name:» выбираем **имя проекта**. Например, Prog1. Сразу под полем с именем проекта расположены два элемента выбора режимов. Так называемые «Чекбоксы» (Check box). По умолчанию оба чек-бокса выбраны (то есть, в соответствующих квадратиках проставлены «галочки»).

Первый чек-бокс (Create initialize file) определяет, нужно ли автоматически создавать главный программный файл. Если у вас уже есть файл с тестом программы на Ассемблере и вы просто хотите создать проект, а затем подключить туда готовый программный файл, снимите соответствующую «галочку». Если вы создаете проект «с нуля», оставьте «галочку» нетронутой.

Второй чек-бокс (Create folder) определяет, нужно ли автоматически создавать отдельный каталог для данного проекта. Если вы заранее уже создали нужный каталог средствами Windows, снимите пометку. Если нет, оставьте. Следующее поле называется «Initial file». Оно должно содержать имя файла, куда будет записываться текст программы. По умолчанию имя файла уже вписано в это

поле. Оно соответствует имени проекта. Советую оставить его без изменений.

Еще одно поле, требующее нашего вмешательства, — это **поле «Location»**. Здесь вы должны указать путь к тому месту на вашем жестком диске, где будет храниться проект. Путь нельзя ввести непосредственно с клавиатуры. Для изменения пути нужно нажать кнопку справа, на которой в качестве названия поставлено многоточие («...»).

Откроется диалог «Select folder», при помощи которого вы и должны выбрать директорию. Просто войдите в нужную директорию и нажмите кнопку «Select». При выборе директории нужно учитывать значение чек-бокса «Create folder». Если там стоит «галочка», то при выборе в качестве Location каталога «c:\AVR\myprog», программа поместит ваш проект в каталог «c:\AVR\myprog\Prog1».

На этом можно закончить работу с первым окном построителя. Но прежде, чем нажимать кнопку «Next>>», обратите внимание, что в нижней части окна имеется **еще один чек-бокс**. Он называется **«Show dialog at startup»**. При выборе этого элемента, диалог создания проекта будет автоматически запускаться каждый раз при запуске программы AVR Studio.

Для перехода **к следующему этапу** построения проекта нажмите кнопку «Next>>». Содержимое окна построителя изменится. Появятся два больших поля под общим названием «Select debug platform and device» (Выбор отладочной платформы и микроконтроллера). В списке Отладочных платформ («Debug platform») перечислены все отладочные платы, которые поддерживает данная программа.

Мы не будем использовать внешних плат, поэтому выберем пункт «AVR Simulator» (Программный имитатор AVR). В поле «Device» выбираем нужный тип микросхемы. В нашем случае это ATiny2313. Теперь все настройки закончены. Для завершения процесса нажмите кнопку «Finish». После нажатия этой кнопки программа создает проект и записывает его в выбранную вами директорию.

Сразу после создания новый проект состоит всего из двух файлов:

- собственно файл проекта `Prog1.aps`;
- файл, куда будет помещен текст программы на Ассемблере `Prog1.asm`.

Файл текста программы автоматически открывается в окне 3. Причем он пока абсолютно пустой. Теперь вы можете **приступать**

к набору этого текста. Если речь идет о программе Prog1, то просто наберите текст, приведенный в **листинге 1.1**. При наборе текста вы можете пользоваться всеми возможностями, какие обычно предоставляет любой современный текстовый редактор.

Встроенный текстовый редактор программы AVR Studio поддерживает все необходимые сервисные функции:

- выделение текстовых фрагментов;
- вырезание;
- копирование;
- вставку;
- перетаскивание мышью;
- поиск и замену и многое другое.

Для управления всеми этими возможностями используется стандартный интерфейс, знакомый вам по многим текстовым редакторам, в частности, по популярному редактору Microsoft Word. Набранный тест программы не забудьте записать на диск при помощи команды «Save» меню «File» или при помощи соответствующей кнопки на панели инструментов (). Кнопка позволяет записать сразу все открытые текстовые файлы.

Для программ, приведенных в этой книге, проекты создавать не обязательно. Достаточно скачать файл с электронными версиями программ с сайта http://book.microprocessor.by.ru, распаковать архив и поместить его содержимое в любую подходящую директорию.

Например, в директорию c:\AVR\myprog\. После распаковки у вас появится целый набор директорий, в каждой из которых помещен свой проект. Причем архив содержит не только проекты на Ассемблере, но и на СИ. Любой проект на Ассемблере можно открыть при помощи пункта «Open Project» меню «Project».

2.1.4. Трансляция программы

Форматы файлов

После того, как текст программы набран и записан на жесткий диск, необходимо произвести **трансляцию программы**. В процессе трансляции создается результирующий файл, который представляет собой ту же программу, но в машинных кодах, предназначенную для записи в программную память микроконтроллера. Результирующий файл имеет расширение hex.

Кроме hex-файла транслятор создает еще несколько вспомогательных файлов. И главное, файл с расширением eep. Этот файл имеет точно такую же внутреннюю структуру, как файл hex. А содержит он информацию, предназначенную для записи в EEPROM. Такая информация появляется в том случае, когда в тексте программы переменным, размещенным в сегменте eeprom, присвоены начальные значение. В наших примерах мы этого не делали. Поэтому файл с расширением eep во всех проектах будет пустой (содержать лишь завершающую строку).

Теперь немного разберемся с форматом файлов hex и eep. В обоих случаях применяется так называемый HEX-формат, который практически является стандартом для записи результатов транслирования различных программ. Он поддерживается практически всеми трансляторами с любого языка программирования.

В принципе, программисту не обязательно знать структуру этого формата. Достаточно понимать, что в hex-файле определенным способом закодирована программа в машинных кодах. Именно этот файл используется программатором для «прошивки» программной памяти микроконтроллера. Любой программатор поддерживает hex-формат и распознает записанные туда коды автоматически. Однако для тех, кому это интересно, приведу краткое описание hex-формата.

Формат HEX-файла

Если вы посмотрите содержимое такого файла при помощи редактора «Блокнот», то вы увидите, что это текстовый файл, в котором данные закодированы в виде текстовых строк. Ниже приведено содержимое hex-файла, полученного в результате трансляции программы Prog1 (**листинг 1.1**):

:020000020000FC

:100000000FE70DBF00E806BD00E006BD01BB0FEF26

:1000100007BB08BB02BB00E808B900B308BBFDCFB3

:00000001FF

Как видите, данный файл состоит из четырех строк. Первая и последняя строки несут служебную информацию. Наличие первой строки необязательно. Система AVR Studio при трансляции программы всегда добавляет в hex-файл первую строку именно такого содержания. Последняя строка — это стандартный конец для любого hex-файла.

Оставшиеся две строки как раз и содержат информацию о кодах программы. В каждой такой строке закодирована цепочка байтов и адрес в памяти, где эти байты должны размещаться.

Строка начинается с двоеточия. Двоеточие — обязательный элемент, который служит для идентификации hex-формата. Все остальные символы в строке — это шестнадцатиричные числа, записанные слитно без пробелов. Отдельные числа отличают по их позиции в строке. Так первые два знака занимает шестнадцатиричное число, означающее длину цепочки.

В нашем случае длина обеих цепочек равна 0x10 (то есть 16) байт. Следующие четыре символа — это начальный адрес, куда эти байты должны быть помещены. **Первая цепочка** будет размещена в памяти, начиная с нулевого адреса. **Вторая цепочка** — с адреса 0x0010. Очередные два знака занимает код вида строки. В интересующих нас строках он равен «00», что означает, что эти строки предназначены для записи данных (в первой строке такой код равен «02», а в последней «01»).

Сразу после кода вида строки начинаются собственно данные. Каждый байт данных занимает два знака. Самые последние два символа — это контрольная сумма. Она рассчитывается по специальной формуле с использованием значений всех байтов цепочки и служит для проверки на отсутствие ошибок.

Процедура трансляции

Но **вернемся к процедуре трансляции**. Для того, чтобы запустить процесс трансляции текущего проекта, нужно выбрать в меню «Build» пункт, который тоже называется «Build», или нажать кнопку . Длительность процесса трансляции зависит от размеров программы.

Сразу же после начала процесса вкладка «Build» в окне 2 выходит на передний план.

В процессе трансляции сюда выводятся служебные сообщения. К таким сообщениям относятся: сообщения о завершении различных этапов трансляции, сообщения об ошибках (Error), а также предупреждения (Warning).

В готовой отлаженной программе ошибок и предупреждений быть не должно. Если программа обнаружит критическую ошибку (Error), то процесс трансляции будет приостановлен, и результирующие файлы созданы не будут. В этом случае необходимо устранить ошибки и повторить трансляцию.

Естественно, транслятор не в состоянии найти все виды ошибок. Он находит только явные ошибки, которые можно найти автоматически. К таким ошибкам относятся:

- ошибки синтаксиса (неправильное написание имени команды);
- неверное количество параметров у оператора;
- попытка использования неописанных переменных и т. п.

Например, сообщение «Unknown instruction or macro» означает, что найдена «Неизвестная инструкция или макрокоманда».

Предупреждения — это тоже ошибки, но не критические. При возникновении некритической ошибки процесс трансляции завершается как обычно. Все результирующие файлы создаются в полном объеме. Однако прежде чем зашивать такую программу в микроконтроллер, тщательно проанализируйте сообщение и постарайтесь определить, как оно повлияет на результаты работы. В любом случае, лучше изменить программу таким образом, чтобы устранить все предупреждения.

Все сообщения во вкладке «Build» появляются по мере их поступления. Для наглядности каждое сообщение помечено цветным кружочком в начале строки:

- сообщения об ошибках помечаются кружочком красного цвета;
- предупреждения — желтым кружочком;
- сообщения об успешном выполнении каждого очередного этапа трансляции помечаются зеленым кружочком.

Если сообщения не вмещаются в окно, то они скрываются в верхней его части. Однако, используя полосу прокрутки, их всегда можно просмотреть. В случае успешного завершения процесса трансля-

ции в качестве последнего сообщения выводится статистическая информация (см. **раздел 2.1.2**).

Каждое сообщение об ошибке во вкладке «Build» содержит точное указание места в программе, где произошла эта ошибка. При этом указывается

- имя файла;
- номер строки;
- фрагмент текста программы, содержащий ошибку;
- ее расшифровка.

Для того, чтобы быстро перейти к фрагменту программы, содержащему эту ошибку, достаточно двойного щелчка по сообщению об ошибке. Окно с текстом программы выйдет на передний план, и в этом окне автоматически отобразится нужный участок текста. На левой границе окна напротив строки, содержащей ошибку, вы увидите синюю стрелочку — указатель ошибки.

Иногда программа неверно определяет место, где возникла ошибка. Это происходит из-за несовершенства анализатора синтаксиса. Дело в том, что очень сложно разработать идеальный алгоритм анализа ошибок. Если в какой-либо строке транслятор показывает ошибку, а вы ошибок не наблюдаете, посмотрите на предыдущие строки. Возможно, ошибка где-то там.

2.1.5. Отладка программы

Ошибки алгоритма и его реализации

Если вы исправили все ошибки и добились отсутствия предупреждений, то это значит, что программа успешно оттранслирована. В принципе, вы можете записывать ее в программную память и пробовать ее работу «в железе». Но в большинстве случаев отсутствие синтаксических ошибок еще не означает отсутствие ошибок как таковых. Можно написать команду правильно, да не ту. Но самая **главная неприятность — ошибки алгоритма или его реализации**.

Программист может упустить какой-либо шаг или неправильно поставить условие. Всех возможных ошибок алгоритма не перечесть. Но в результате программа может работать неправильно либо совсем не работать. По этой причине перед тем, как записывать

программу в программную память микроконтроллера, необходимо попытаться выявить все эти ошибки.

Вообще, процесс написания программы процентов на 60—70 состоит из поиска и устранения ошибок. И основное количество ошибок выявляется при отладке программы. Все программные примеры, приведенные в этой книге, прежде чем появились на ее страницах, прошли процесс отладки.

И несмотря на простоту этих программ и достаточный опыт в программировании, мне пришлось исправить немало ошибок. По этому поводу существует народная программистская шутка: «*Если ты написал программу и транслятор не обнаружил в ней ни одной ошибки, посмотри, все ли в порядке с транслятором!*».

С большим юмором подошли к этому вопросу англичане. По-английски процесс отладки называется Debug (Дебаг). Слово «Bug» — означает блоха. А «Debug» — это процесс избавления от ошибок или процесс ловли блох. Именно этим вам и придется заняться.

Этапы процесса отладки

Процесс отладки начинается с перевода программы в соответствующий режим. Если проект открыт, а все его программы записаны и оттранслированы, то для перехода в режим отладки выберите пункт «Start Debugging» в меню «Debug» или нажмите кнопку [▶] на панели задач.

Программа начнет **процесс подготовки**. Процесс длительный. Пока идет подготовка, в нижней части основной панели будет двигаться полоса, показывающая процент выполнения операции. По окончании процесса подготовки программа переходит **в новый режим**. В окне 1 на передний план выходит вкладка «I/O View» (см. **рис. 2.4**), которая теперь будет использоваться для просмотра содержимого всех регистров. Причем внешний вид этой вкладки немного изменяется. Для каждого элемента в дереве ресурсов появятся поле, отображающее его содержимое.

В окне 2 на передний план выходит вкладка «Breakpoints and Tracepoints», где теперь будут отображаться все точки останова. В панели инструментов активизируются все инструменты, относящиеся к режиму отладки (до этого они были неактивны). В окне 3 на первый план выходит текст главного программного файла. На левой

границе окна этого файла появляется желтая стрелка — указатель текущей выполняемой команды. Указатель установится в начало программы (напротив первой исполняемой команды). Теперь все готово для отладки.

Отладка может выполняться **разными методами**. Самый простой метод — **пошаговое выполнение**. Для того, чтобы сделать один шаг, выберите в меню «Debug» пункт «Step into» («Шаг в») либо нажмите кнопку [] на панели инструментов.

Можно также просто нажать кнопку «F11». В результате программа выполнит одну текущую команду. Указатель текущей команды (желтая стрелка) переместится в следующую позицию. Содержимое регистров изменится в соответствии с выполненной операцией.

Вы можете это проверить, найдя нужный регистр в окне 1. Убедившись, что команда выполнена правильно, делайте следующий шаг. И так далее. При этом вы можете проследить последовательность выполнения операций, правильность выполнения условных переходов и многое другое.

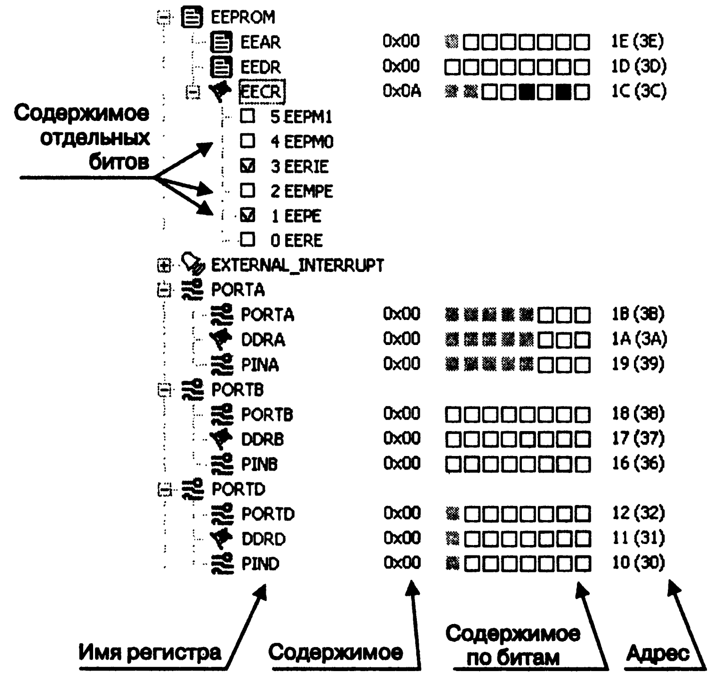


Рис. 2.4. *Изменение содержимого регистров*

В любой момент вы можете **вручную изменить** содержимое любого из элементов в дереве ресурсов. Причем можно изменять как содержимое любого отдельного разряда, так и всего регистра в целом. Для изменения содержимого разряда достаточно щелкнуть при помощи мыши по одному из квадратиков, символизирующему нужный разряд (см. **рис. 2.4**).

При этом состояние квадратика изменится на противоположное (единица изменится на ноль либо наоборот). Для изменения значения всего регистра необходимо произвести двойной щелчок мышью по изображению содержимого регистра (в шестнадцатиричном виде). Откроется окно содержимого. В этом окне вы можете выбрать одну из четырех форм представления числа (шестнадцатиричное, десятичное, восьмиричное или двоичное) и изменить это значение в выбранном вами формате. Затем нажмите кнопку «Ok» и изменение «запишется» в соответствующий регистр.

Изменяя содержимое регистра, вы можете моделировать различные ситуации. **Например**, имитировать изменение сигналов на входе порта или принудительно изменять значение счетного регистра таймера, чтобы ни ждать, пока он досчитает до нужного значения.

Кроме директивы «Шаг в», имеется еще несколько ее модификаций. Их назначение и способы вызова приведены в **табл. 2.1**.

Таблица 2.1. *Директивы пошагового выполнения программы*

Название	Пункт меню «Debug»	Кнопка	Горячая клавиша	Описание
Шаг в	Step into	{↓}	F11	Выполнить очередную команду
Шаг через	Step over	{}↓	F10	Выполнить очередную подпрограмму
Шаг из	Step out	{}↓	Shift+F11	Завершить текущую подпрограмму
Выполнить до	Run to cursor	→{}	Ctrl+F10	Выполнять с текущей строки и до строки, где стоит курсор

Директива «Шаг через» используется в том случае, если при пошаговом выполнении программы встретится команда вызова подпрограммы. Если вы не хотите пошагово выполнять всю подпрограмму, вы можете выполнить ее за один шаг. При этом желательно, чтобы подпрограмма не содержала ошибок.

Директива «Шаг из» применяется в том случае, если вы все же вошли в подпрограмму, но затем поняли, что ее пошаговое выполнение излишне. Выбрав данную директиву, можно за один шаг выполнить все оставшиеся команды подпрограммы.

Директива «Выполнить до» применяется в том случае, когда какая-либо часть программы не оформлена в виде подпрограммы, но ее желательно выполнить за один шаг. В этом случае в конце выбранного фрагмента вы можете установить текстовый курсор (мигающую вертикальную полоску) и выбрать директиву «Выполнить до». Отладчик за один шаг выполнит все команды, начиная с текущей (отмеченной желтой стрелкой) и вплоть до текстового курсора. Команда в строке с курсором выполняться не будет. Она станет текущей (на нее теперь будет указывать желтая стрелка).

Применение точек останова

Пошаговый метод отладки удобен для отладки небольших несложных программ или отдельных участков большой программы. Но представьте себе, что ваша программа содержит цикл, который должен быть выполнен большое количество раз. Для того, чтобы проверить правильность выполнения всего этого цикла в пошаговом режиме, вам пришлось бы очень долго щелкать мышкой! В подобных случаях применяются **точки останова (Breakpoint)**.

Определение. *Точка останова — это специальная метка, которую в отладочном режиме программист может поставить против любой строки программы.*

Затем программа запускается под управлением отладчика. Но это — не реальная работа. Это лишь имитация работы микроконтроллера. Программа выполняется строка за строкой, пока в очередной строке не встретится точка останова. Обнаружив такую точку, отладчик приостанавливает выполнение программы.

Выглядит это таким образом, как-будто за один шаг вы выполнили большой кусок программы. Теперь вы можете снова просмотреть и (или) изменить содержимое любого регистра. А затем продолжить отладку. Причем, вы можете продолжить ее как в пошаговом

режиме, так и запустить программу в режиме автоматического выполнения до следующей точки останова.

Для управления точками останова программа имеет несколько встроенных директив, которые показаны в **таблице 2.2.**

Таблица 2.2. *Директивы управления точками останова*

Название	Пункт меню «Debug»	Кнопка	Горячая клавиша	Описание
Поставить точку останова	Toggle Breakpoint		F9	Поставить (снять) точку останова в строке, где находится курсор
Убрать все точки останова	Remove all Breakpoints		-	Убрать все поставленные ранее точки останова
Создать программную точку останова	New Breakpoints / Program Breakpoint	-	-	Создать точку останова путем задания программного условия
Создать точку останова по данным	New Breakpoints / Data Breakpoint	-	-	Создать точку останова путем задания условия по данным

Для того, **чтобы поставить точку останова** в какой-либо строке программы, нужно сначала поместить в эту строку текстовый курсор. Затем выбрать директиву «Поставить точку останова» (см. **табл. 2.2**). Точка останова выглядит как коричневый кружочек напротив выбранной строки программы на левой границе текстового окна.

Если поместить курсор в строку, где уже есть точка останова, и выполнить еще раз директиву «Поставить точку останова», то точка убирается. Убрать сразу все поставленные точки останова можно при помощи директивы «Убрать все точки останова».

Второй способ простановки точек останова — задание их через меню. Предназначенный для этого пункт «New Breakpoints» меню «Debug» имеет два подпункта. При помощи подпункта «Program Breakpoint» можно устанавливать программные точки останова. То есть точно такие, какие мы ставили предыдущим способом.

Отличие способа постановки точек через меню в том, что их местоположение в программе вы определяете путем заполнения полей в специальной форме. В этой форме, кроме номера строки или адреса программы, где вы хотите поставить точку останова, вы можете указать количество проходов.

Для этого вам необходимо заполнить поле «Break execution after: — hits» («Остановить выполнение после: — проходов»). Если число в этом поле не равно нулю, то программа остановится в данной точке останова не с первого раза, а лишь тогда, когда пройдет через нее указанное количество раз.

Если вы установили вашу точку останова не через меню, а напрямую в тексте программы, вы все равно можете вызвать описанный выше диалог и изменить в нем количество проходов, щелкнув мышью по строке с описанием нужной точки останова во вкладке «Breakpoints and Tracepoints».

При помощи подпункта «Data Breakpoint» пункта «New Breakpoints» меню «Debug» можно задавать точки останова по данным. При выборе этого пункта меню открывается диалог, в котором вы можете выбрать любую из переменных вашей программы или любой ресурс микроконтроллера (из открывающегося списка) и поставить точку останова по обращению к этой переменной (ресурсу).

Программа позволяет выбрать целый **ряд условий**, при которых наступит останов программы. По умолчанию останов происходит при любом обращении к этой переменной как в режиме чтения, так и в режиме записи. Вы можете выбрать другое условие. **Например**, при равенстве переменной определенному значению. Выбор условия производится при помощи поля «Break when:» («Остановиться если:») и поля «Access type:» («Тип доступа»). Имя переменной выбирается при помощи поля «Location».

Диалог простановки точек останова обоих видов можно вызывать не только через меню. В верхней левой части вкладки «Breakpoints and Tracepoints» для этого имеется специальная кнопка.

После того, как вы проставили все точки останова, вы можете запускать программу в **режиме автоматического выполнения**. Для управления отладчиком в этом режиме программа AVR Studio также имеет несколько специальных директив (см. **табл. 2.3**). Запуск автоматического выполнения программы производится при помощи директивы «Пуск».

Пока программа находится в режиме автоматического выполнения, новое состояние регистров не отображается. Указатель текущей команды также отсутствует. В нижней строке главной панели программы в правой ее стороне находится индикатор состояния. В режиме останова это желтый кружочек с минусом посередине. Слева от него находится слово «Stopped» (Остановлено). В режиме автоматического выполнения программы желтый кружочек

превращается в зеленый с плюсом внутри. Вместо слова «Stopped» появляется слово «Running» (Запущено).

Таблица 2.3. *Директивы управления процессом отладки*

Название	Пункт меню «Debug»	Кнопка	Горячая клавиша	Описание
Запустить	Run		F5	Запуск автоматического выполнения программы с текущей команды
Остановить	Break		Ctrl+F5	Остановка автоматического выполнения программы
Сброс	Reset		Chift+F5	Исходное состояние (сброс микроконтроллера)
Закончить отладку	Stop Debugging		Ctrl+Chift+F5	Закончить отладку.

Если вы неправильно поставили точку останова либо и вовсе забыли ее поставить, программа будет находиться в режиме автоматического выполнения бесконечно долго. Для досрочной остановки программы используется директива «Остановить». Если в процессе отладки программы понадобится начать все сначала (сымитировать сброс микроконтроллера), это можно сделать при помощи директивы «Сброс». По окончании отладки программы необходимо перейти в режим редактирования. Для этого служит директива «Закончить отладку».

Просмотр и изменение содержимого введенных переменных

Для оперативного просмотра и изменения содержимого введенных вами переменных в процессе отладки можно открыть специальное окно. Для этого достаточно выбрать пункт Watch в меню View. Окно имеет четыре вкладки. Поэтому можно иметь четыре разных набора переменных.

Для того чтобы включить какую-либо переменную в текущее окно Watch, необходимо установить курсор мыши на имя этой переменной в тексте программы и нажать правую кнопку мыши. Допустим, вы установили курсор на переменную temp. Тогда в открывшемся

меню вы увидите пункт Add Watch: «temp». Выберите этот пункт, и переменная будет включена в список Watch.

Точно так же можно оперативно просматривать содержимое любого вида памяти. Для этого выберите пункт «Memory» в меню «View». Откроется новое окно под названием «Memory». По умолчанию в этом окне в виде дампа будет представлено содержимое программной памяти. При помощи выпадающего списка в левой верхней части этого окна можно выбрать другой вид памяти. Память данных (Data), EEPROM или даже содержимое РОН или портов вода/вывода. В процессе отладки вы всегда будете видеть в этом окне все изменения выбранной части памяти. Если вы желаете видеть одновременно содержимое сразу нескольких видов памяти, то вы можете открыть второе и даже третье подобное окно. Для этого выберите пункт «Memory2» или «Memory3» в меню «View».

2.1.6. Исправление ошибок

Все программы, приведенные в данной книге, уже отлажены и изменения в них не требуется. Однако в том случае, если вы захотите доработать программу либо написать новую, вам придется много раз переписывать ее, искать различные фрагменты, заменять их на другие и т. д. Редактор программы AVR Studio дает полный спектр стандартных средств редактирования. Одно из таких средств — это простановка закладок. Поставив закладку в любом месте в тексте программы, вы можете спокойно листать этот текст дальше. В случае необходимости вы можете в любой момент вернуться к закладке. В **табл. 2.4** приведены все директивы работы с закладками.

Для создания новой закладки нужно установить в нужной строке текстовый курсор и выбрать директиву «Поставить закладку». При повторном вызове этой директивы в той же **строке**, закладка убирается. Проставив несколько закладок, можно передвигаться по ним при помощи директив «Перейти к следующей закладке» и «Перейти к предыдущей закладке». При помощи соответствующей директивы можно убрать все закладки.

Таблаца *2.4.* *Директивы работы с закладками*

Название	Пункт меню «Edit»	Кнопка	Горячая клавиша	Описание
Поставить закладку	Toggle Bookmark		Ctrl+F2	Поставить (снять) закладку в строке, где находится курсор
Перейти к следующей закладке	-		F2	Переместить курсор к следующей строке с закладкой
Перейти к предыдущей закладке	-		Chift+F2	Переместить курсор к предыдущей строке с закладкой
Убрать все закладки	Remove Bookmarks		Ctrl+Chift+F2	Удалить все поставленные ранее закладки

2.1.7. Создание проектов на языке СИ

Как уже упоминалось ранее, программа AVR Studio позволяет создавать, транслировать и отлаживать проекты на языке СИ. При этом для трансляции используется программный продукт стороннего производителя под названием **WinAVR**, который в случае установки на ваш компьютер автоматически интегрируется с программной средой AVR Studio.

Программа WinAVR представляет собой набор утилит, предназначенных для разработки программ для микроконтроллеров AVR. Она включает в себя простейшие средства разработки, в том числе компилятор с языков С и С++. Компилятор имеет открытую лицензию, так называемую GNU (General Public License). Открытая лицензия предполагает распространение программ в полностью доступном виде вместе с исходным текстом и разрешает не только любое некоммерческое использование программы, но и доработку текста программы по своему усмотрению. Отрицательным моментом такого способа предоставления лицензий является отсутствие каких-либо гарантий работоспособности программы. Все ошибки исправляйте сами!

Программу WinAVR можно свободно скачать с сайта производителя по адресу **http://winavr.sourceforge.net/download.html.** После установки этой программы на ваш компьютер программа AVR Studio приобретает возможность транслировать программы с языка СИ.

При этом процессы создания проекта, трансляции всех его программ, а также процесс их отладки будут выглядеть точно так же, как и в случае программ на Ассемблере. Отличие будет только в самом начале. При создании проекта вы должны выбрать другой тип проекта. Вместо пункта «Atmel AVR Assembler» нужно выбрать «AVR GCC».

Несмотря на очевидные преимущества бесплатных условий распространения данной программы, для начинающих программистов я бы не рекомендовал использовать WinAVR. Именно по этой причине все программные примеры в данной книге выполнены при помощи другой программной среды, которая называется «**Code Vision AVR**». Подробнее о CodeVisionAVR речь пойдет в следующем разделе.

Система WinAVR и система Code Vision AVR поддерживают разные версии языка СИ. Поэтому, если все же вы решите попробовать применить WinAVR, то учтите, что программные примеры, приведенные в **главе 4** данной книги, не пригодны для системы WinAVR без определенной переработки. И хотя необходимая доработка не носит кардинального характера, без определенных знаний выполнить ее невозможно.

Для тех, кто хочет научиться этому самостоятельно, в файле программных примеров, который я уже рекомендовал вам скачать с моего сайта в Интернете, имеется несколько проектов, представляющих собой уже знакомые нам программы, переделанные под GCC.

Ниже описана другая, система программирования, позволяющая создавать программы на языке СИ. Это программная среда Code Vision. В отличии от WinAVR, система Code Vision гораздо удобнее, компактнее и работает более устойчиво.

2.2. Система программирования Code Vision AVR

2.2.1. Общие сведения

С системой Code Vision AVR мы уже немного знакомы. В первой **главе (раздел 1.2)** подробно рассматривалась работа с мастером-построителем проектов. Теперь настал момент познакомиться с программой Code Vision AVR подробнее. Эта программа разработана румынской фирмой «HP Infotech», специализирующейся на разработке программного обеспечения.

Инсталляционный пакет свободно распространяемой версии программы, рассчитанной на создание программ, результирующий код которых не превышает 2 Кбайт, можно скачать в Интернете по адресу **http://www.hpinfotech.ro/html/download.htm.** Там же можно скачать архив с полной и облегченной коммерческими версиями той же программы, защищенные паролем. Эти версии платные. Условия предоставления права на использование этих программ можно прочитать на той же самой странице.

Как по назначению, так и по структуре и устройству, программа Code Vision AVR очень напоминает AVR Studio. Главное **отличие** — отсутствие собственных средств отладки. Для отладки программ Code Vision AVR пользуется отладочными средствами системы AVR Studio.

Система Code Vision AVR, так же как AVR Studio, работает не с программами, а с проектами. В **разделе 1.2** мы достаточно подробно рассматривали процесс создания проекта, формирования заготовки будущей программы и превращения этой заготовки в законченную программу. Что же еще может система Code Vision? Она может проверять программу на наличие синтаксических ошибок, производить трансляцию программы и сохранение результатов трансляции в HEX-файле, а также производить расширенную трансляцию.

В процессе расширенной трансляции программы формируется не только HEX-файл, но и файл той же программы, переведенный на язык Ассемблер, а также специальный файл в COF-формате, предназначенный для передачи программы в AVR Studio для отладки. После создания и расширенной трансляции проекта его директория будет содержать файлы со следующими расширениями:

prj — файл проекта Code Vision AVR;

txt — файл комментариев. Это простой текстовый файл, который вы заполняете по своему усмотрению;

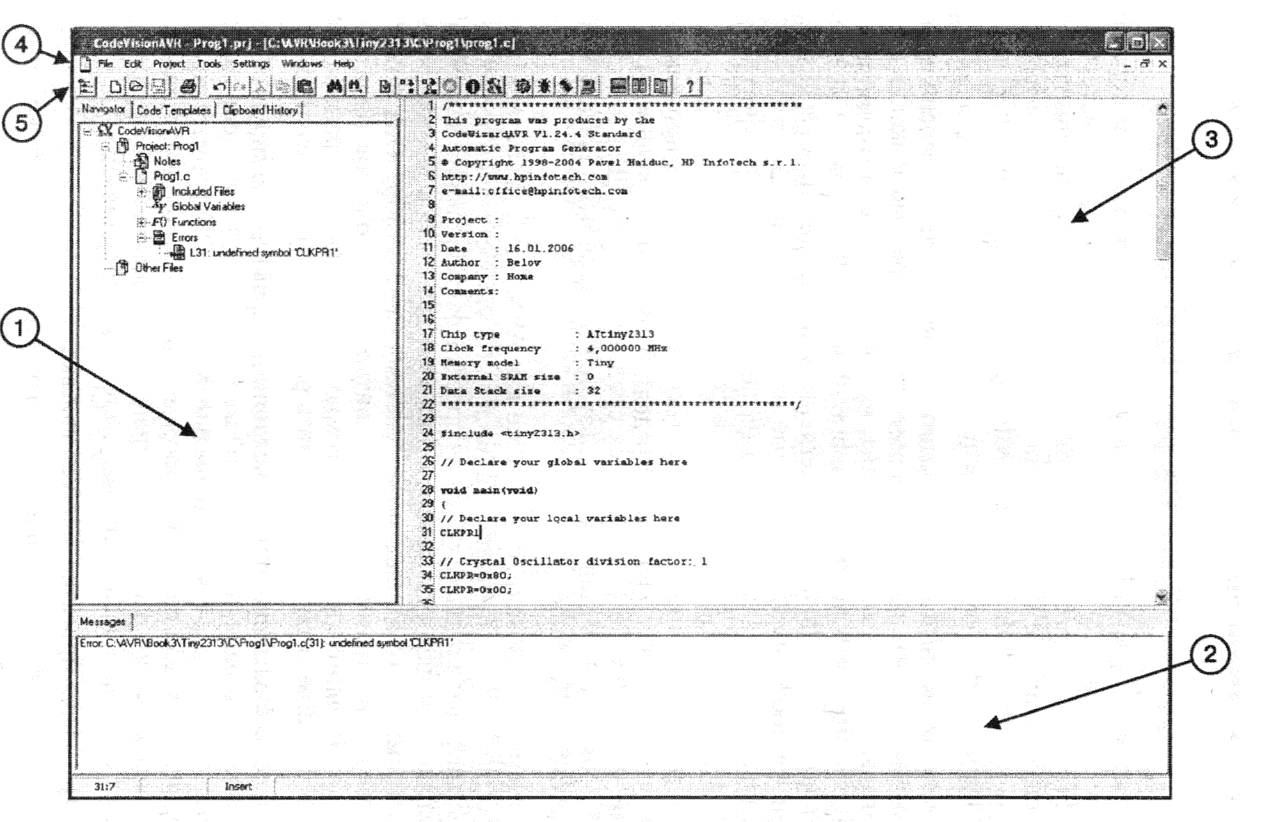


Рис. 2.5. *Основное окно программы Code Vision*

c — текст программы на языке СИ;

asm — текст программы на Ассемблере (сформирован Code Vision);

cof — формат для передачи программы в другие системы для отладки;

eep — содержимое EEPROM (формируется одновременно с HEX-файлом);

hex — результат трансляции программы;

inc — файл-дополнение к программе на Ассемблере с описанием всех зарезервированных ячеек и определением констант;

lst — листинг трансляции программы на Ассемблере;

map — распределение памяти микроконтроллера для всех переменных программы на СИ;

obj — объектный файл (промежуточный файл, используемый при трансляции);

rom — описание содержимого программной памяти (та же информация, что и в HEX-файле, но в виде таблицы);

vec — еще одно дополнение к программе на Ассемблере содержащее команды переопределения векторов прерываний;

cwp — файл построителя проекта. Содержит все параметры, которые вы ввели в построитель (см. **раздел 1.2**).

Все перечисленные выше файлы имеют одинаковые имена, соответствующее имени проекта. Кроме перечисленных выше файлов, директория проекта может содержать несколько файлов с расширением типа `c~`, `pr~` или `cw~`. Это страховочные копии соответственно файлов `c`, `prj` и `cwp`. То же самое, что файл `bak` для текстовых файлов.

2.2.2. Интерфейс системы Code Vision AVR

Окно номер 1

Интерфейс программы Code Vision AVR показан на **рис. 2.5**. На первый взгляд он напоминает интерфейс AVR Studio. Но здесь он гораздо проще. Основная панель Code Vision тоже разделена на три окна. **Окно номер 1** имеет три вкладки разного назначения. «Корешки» этих вкладок расположены в верхней части окна.

Первая вкладка называется «Navigator». В окне этой вкладки показана структура текущего открытого проекта. Структура включает

в себя список всех файлов, из которых состоит проект, а также список найденных ошибок и предупреждений, который появляется здесь после трансляции программы. В данном случае под файлами проекта понимаются не все те файлы, которые были перечислены в предыдущем разделе, а только исходные файлы (тексты программ на языке СИ плюс файл описания).

Если файл с текстом программы содержит больше, чем одну функцию, соответствующая этому файлу ветвь в структурном дереве проекта получает разветвление. В основании этой ветви появляется квадратик с плюсиком внутри. Двойным щелчком мыши можно раскрыть соответствующую ветвь и увидеть все функции входящие в данный файл программы. Такая структура очень удобна для навигации. Щелчок мышью по имени любой из функций в дереве проекта приведет к тому, что окно с текстом программы, содержащей эту функцию, переместится на передний план, и текстовый курсор установится в начало выбранной функции.

Вторая вкладка окна 1 называется «Code Templates» (Шаблоны Кода). Эта вкладка задумана как помощь программисту. Она содержит шаблоны нескольких основных конструкций языка СИ. В частности, шаблоны операторов `for`, `while`, `if` и так далее. В любой момент программист может открыть эту вкладку, выбрать нужную структуру и просто «перетянуть» ее при помощи мыши в основной текст программы (в окно 3).

При этом перетянутый текст скопируется. Это значит, что в вашу программу «перетянется» его копия, а оригинал останется в окне «Code Templates». Затем вам нужно лишь заполнить полученный таким образом шаблон командами, и фрагмент программы готов. При желании вы можете пополнить набор шаблонов.

Для того, чтобы ввести новый элемент в список шаблонов, сначала наберите его текст в текстовом окне программы (окно 3), выделите его и «перетяните» при помощи мыши в окно шаблонов. Записанный таким образом шаблон останется в окне шаблонов даже после загрузки нового проекта. С этого момента вы всегда можете использовать его в любой другой программе. Лишние шаблоны можно удалить. Для этого достаточно щелкнуть по ненужному шаблону правой кнопкой мыши и в появившемся меню выбрать пункт «Delete».

Третья вкладка окна номер 1 называется «Clipboard History» (История значений буфера обмена). Ее структура такая же, как структура вкладки «Code Templates». Но вместо шаблонов вкладка содержит все, что в процессе редактирования попадало в буфер обмена. Если вам снова

понадобилось какое-либо из этих значений, вы в любой момент можете «перетянуть» его при помощи мышки в вашу программу.

Окно номер 1 при необходимости можно быстро убрать с экрана. Для этого достаточно нажать кнопку на панели инструментов. При повторном нажатии на эту кнопку окно появляется вновь.

Окно номер 2

Окно номер 2 содержит всего одну вкладку. Она называется **«Messages» (Сообщения)**. Сюда выводятся все сообщения об ошибках и предупреждения в процессе трансляции. Так же, как и в AVR Studio, вы можете быстро перейти к тому месту программы, где найдена ошибка. Для этого достаточно выполнить двойной щелчок мышкой по соответствующему сообщению об ошибке в окне 2.

Окно номер 3

Окно номер 3 так же, как и аналогичное окно в AVR Studio, может содержать одно или несколько окон с текстом программ, в данном случае на языке СИ. Кроме того, там же появляется окно «Project Notes» — окно комментариев к текущему проекту. Все окна, появляющиеся в окне 3, обладают свойствами текстового редактора. Так же, как и в AVR Studio, здесь поддерживаются функции выделения фрагментов, их перетаскивания, копирования, вставки, поиска, поиска и замены и т. д.

Так как система Code Vision AVR не поддерживает функцию отладки, в программе отсутствуют все команды, связанные с этим режимом. Отсутствует в ней также и механизм закладок.

Создание проекта без использования мастера

Программа Code Vision AVR так же, как и любая современная программа, работающая в среде Windows, имеет **строку меню (4)** и **панель инструментов (5)**. При помощи меню можно управлять всеми функциями системы. Кнопки панели инструментов дублируют самые важные команды меню.

Так как процесс создания проекта при помощи мастера подробно описан в **разделе 1.2**, рассмотрим случай создания проекта без использования мастера. Для того, чтобы **начать процесс создания**

проекта, выберите пункт «New» меню «File» или нажмите кнопку [D] на панели инструментов. На экране появится окно «Create new file» (рис. 2.6). В окне предлагается выбрать вид создаваемого файла: Исходный текст программы («Source») или файл проекта («Project»).

Сначала выберите пункт «Source» и нажмите кнопку «Ок». На экране появится текстовое окно. Внесите в это окно текст программы. Можно сначала внести лишь сокращенный вариант, если полного еще не существует. Теперь нужно записать набранный текст на жесткий диск. Для этого выберите пункт «File Save» меню «File» или нажмите кнопку [H]. Откроется диалог записи файла. Вы должны набрать имя файла, выбрать директорию для проекта и нажать кнопку «Сохранить».

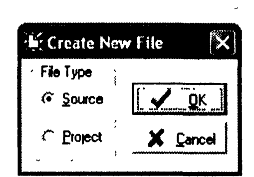


Рис. 2.6. *Создание проекта*

Следующий этап — создание самого проекта. Снова выберите «New» меню «File» или нажмите кнопку [D]. Теперь вместо «Source» выберите «Project» (см. **рис. 2.6**). На вопрос, будете ли вы создавать новый проект при помощи построителя, ответьте «No». Откроется диалог записи файла. Вы должны выбрать имя для файла проекта и записать его в ту же директорию, что и файл с текстом программы. После нажатия кнопки «Сохранить» появляется окно проекта, в котором пока нет файлов с программой (**рис. 2.7**).

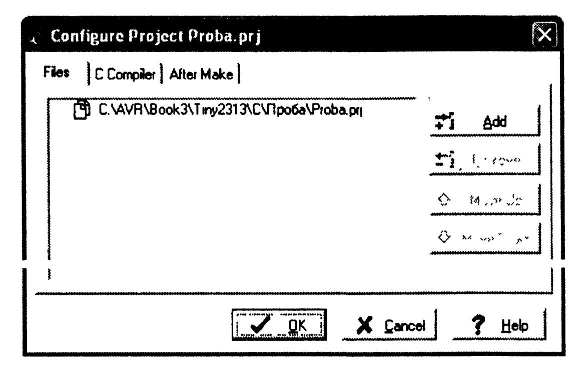


Рис. 2.7. *Окно конфигурации проекта*

Для того, чтобы добавить созданный нами файл к нашему проекту, нажмите кнопку «Add». Откроется диалог, в котором вы должны найти и выбрать созданный ранее файл с программой. После нажатия кнопки «Открыть» файл присоединится к проекту. Теперь нажмите кнопку «Ок», и окно проекта закроется. С вновь созданным проектом вы можете работать так же, как и с проектами, созданными при помощи построителя.

Перед тем, как приступать к трансляции проекта, необходимо **настроить параметры компилятора**. Снова открываем окно «Configure project». Для этого нажимает кнопку . В окне проекта (см. **рис. 2.7**) открываем вкладку «C Compiler». На этой вкладке имеется множество параметров, определяющих стратегию компиляции. Установим только главные из них.

Во-первых, нужно выбрать тип микросхемы. Для этого служит выпадающее меню с заголовком «Chip:». Затем нужно выбрать частоту тактового генератора. Данные о частоте будут использованы транслятором при формировании процедур задержки. Выбор частоты производится при помощи другого выпадающего меню, озаглавленного «Clock:». На этом можно было бы и остановиться. Для остальных параметров можно оставить значения по умолчанию. Однако, при желании, вы можете **выбрать способ оптимизации**. Для выбора способа оптимизации служит поле «Optimize for:» («Оптимизировать по:»). Предлагаются два способа:

- оптимизация по размеру («Size»);
- оптимизация по скорости («Speed»).

Оптимизация по размеру заставляет компилятор создавать результирующий код программы, минимальный по длине. Оптимизация по скорости позволяет создать более длинную, но зато более быстродействующую программу.

После того, как все параметры установлены, запишите все эти изменения, нажав кнопку «Ok» в нижней части окна «Configure project». Окно проекта закроется. Теперь можно **приступать к компиляции**. Директивы режима компиляции приведены в **табл. 2.5.** Если программа достаточно большая, то перед компиляцией можно проверить ее на ошибки. Для этого служит директива «Проверка синтаксиса». При выборе директивы «Компиляция» проверка синтаксиса производится автоматически.

В процессе компиляции создается объектный файл в HEX-формате, а также файл, содержащий данные для EEPROM (файл с расширением eep). Директива «Построить проект» запускает процедуру

полного построения проекта. Полное построение включает в себя проверку синтаксиса, компиляцию, создание файла программы на Ассемблере и файла в формате COF (для передачи в AVR Studio для отладки).

Учтите, что файл COF создается только в том случае, если выставлен соответствующий параметр в окне проекта (окно «Configure project», вкладка «C Compiler», параметр «File output format(s):»). По умолчанию значение этого параметра равно «COF ROM HEX EEP». То есть то, что нам нужно. Однако не мешает все же лишний раз в этом убедиться.

Таблица 2.5. *Директивы работы с программой Code Vision AVR*

Название	Пункт меню «Project»	Кнопка	Горячая клавиша	Описание
Проверка синтаксиса	Check Sintax		-	Проверить синтаксис программы в текущем окне
Компиляция	Compile		F9	Скомпилировать программу
Построить	Make		Shift+F9	Построить проект
Конфигурация	Configure		-	Открыть окно конфигурации проекта

Кроме основных перечисленных выше директив, программа «Code Vision AVR» имеет несколько дополнительных. Некоторые из них перечислены в **табл. 2.6**. **Директива «Запуск мастера»** позволяет в любой момент в процессе редактирования программы запускать мастер-построитель программ. Это может понадобиться для редактирования созданной ранее заготовки программы. Подробно процесс редактирования описан в **разделе 1.2**.

Директива «Отладчик» предназначена для вызова внешней программы-отладчика. При первой попытке вызова этой директивы программа запрашивает путь к исполняемому файлу отладчика. Если указать путь к программе AVR Studio, то в дальнейшем директива «Отладчик» может использоваться для вызова этой программы.

Таблица 2.6. *Дополнительные директивы Code Vision AVR*

Название	Пункт меню «Tools»	Кнопка	Горячая клавиша	Описание
Запуск мастера	CodeWizardAVR		Shift+F2	Запуск мастера-построителя программ
Отладчик	Debugger		Shift+F3	Запуск внешней программы-отладчика
Программатор	Chip Programmer		Chift+F4	Запуск программатора микросхем

Особое значение имеет **директива «Программатор»**. Дело в том, что программа Code Vision AVR имеет в своем составе систему управления программатором. Поддерживается несколько видов программаторов. Благодаря этому программа Code Vision AVR является полнофункциональной. То есть позволяет проводить полный цикл работ по разработке программ для микроконтроллеров.

Действительно, при помощи этой системы можно написать программу, оттранслировать ее и прошить в программную память микроконтроллера. Отсутствует лишь возможность отладки. Но она реализуется при помощи AVR Studio. Перед тем, как пользоваться программатором, необходимо настроить его параметры. Окно настройки параметров программатора открывается при выборе пункта «Programmer» меню «Settings».

Прежде всего, нужно знать название платы программатора, которая подключена к вашему компьютеру. Кроме вида программатора, необходимо выбрать имя порта, к которому он подключен, а также некоторые специальные параметры. Подробнее о программаторах мы поговорим в следующем разделе.

Отладка программы

После того, как программа полностью оттранслирована, можно производить ее отладку. Для этого, не закрывая CodeVisionAVR, необходимо открыть программу AVR Studio. Затем в AVR Studio необходимо выбрать пункт «Open File» в меню «File». Появится диалог открытия файла. В этом диалоге нужно изменить тип открываемого файла. В поле «Тип файла» нужно выбрать «Object Files (*.hex;*.d90;*.a90;*.r90;*.obj;*.cof;*.dbg;)». Затем нужно найти на вашем диске директорию с проектом на CodeVisionAVR.

Когда вы откроете эту директорию, то вы увидите три файла с одинаковым именем. Чтобы узнать, какой из этих файлов имеет расширение «cof», нужно подвести к каждому файлу курсор мыши и посмотреть его описание во всплывающем желтом окне подсказки. Обычно это самый первый (верхний) из файлов. Выберите его и нажмите кнопку «Открыть». Сразу после этого откроется другой диалог. На сей раз диалог записи файла.

Программа предложит записать на диск файл проекта в формате AVR Studio. Имя этого файла уже будет дано по умолчанию. Просто нажмите кнопку «Сохранить», и файл проекта сохранится в той же директории, что и проект на СИ. После этого откроется знакомое нам окно выбора микросхемы и отладочной платформы. Выберите платформу «AVR Simulator», а в качестве микросхемы — ту, которую вы используете в своей программе. Нажмите кнопку «Finish».

После нажатия этой кнопки начнется процесс подготовки к режиму отладки. Как только он закончится, программа AVR Studio перейдет в отладочный режим. При этом в окне номер 3 вы увидите текст программы на языке СИ, а содержимое остальных окон будет точно такое же, как и при отладке программ на Ассемблере.

В процессе отладки вы можете пользоваться всеми удобствами и преимуществами программы AVR Studio:

- ставить точки останова любого типа;
- просматривать и изменять содержимое всех регистров;
- запускать программу в пошаговом режиме и в режиме выполнения под управлением отладчика.

Если в процессе отладки обнаружится ошибка, исправлять ее нужно следующим образом:

- не закрывая AVR Studio, перейдите в окно CodeVisionAVR и измените текст программы;
- запишите изменения и перетранслируйте программу;
- вернитесь в AVR Studio, где вы увидите сообщение о том, что программа изменилась, и предложение учесть изменения;
- ответьте «Yes» и продолжайте отладку;
- по окончании отладки закройте программу AVR Studio.

2.3. Программаторы

2.3.1. Общие сведения

Итак, мы научились создавать схемы на микроконтроллерах, писать программы для них, а также компилировать и отлаживать эти программы. Теперь нам остается заключительный этап — записать оттранслированную программу в программную память микроконтроллера и опробовать ее работу на практике. Для записи программного кода в память микроконтроллера используются специальные устройства — **программаторы**.

Программатор подключается к компьютеру и управляется специальной программой. Микросхема микроконтроллера, в свою очередь, подключается к программатору. Любой микроконтроллер имеет специальный режим — **режим программирования**. В этом режиме все или несколько выводов микросхемы меняют свои функции. В новом режиме они принимают данные и сигналы управления от программатора. Включение режима программирования производится при помощи входа Reset.

Как вы знаете, в рабочем состоянии на этом входе должна присутствовать логическая единица. Подача нулевого потенциала на этот вход приведет к сбросу микроконтроллера. Если точнее, для сброса нужно подать низкий логический уровень на вход Reset на короткое время, а затем опять перевести этот вход в единичное состояние. Если же подать и удерживать ноль, то микроконтроллер перейдет в режим программирования.

Подробнее о режимах программирования микросхем серии AVR вы можете узнать из специальной литературы. Но эти подробности могут пригодиться только в том случае, если вы желаете разработать свой собственный программатор. Однако нам вовсе не обязательно изобретать велосипед. В настоящее время разработано и успешно используется огромное количество различных программаторов. Достаточно выбрать из них подходящий и научиться использовать его для своей работы.

Как вы уже знаете, микросхемы AVR поддерживают несколько способов программирования. Основные из них:

- параллельное программирование;
- программирование по последовательному ISP-каналу.

Причем некоторые модели поддерживают лишь один из этих способов, но большинство поддерживают оба **[3]**. **При параллельном**

программировании микросхему микроконтроллера обычно вынимают из панельки платы, где она должна работать, и вставляют в панельку программатора. После программирования ее необходимо извлечь из программатора и вставить в рабочую схему.

Последовательное программирование не требует обязательного извлечения микросхемы из отлаживаемой схемы. Канал ISP, используемый в этом случае, разработан таким образом, что позволяет производить так называемое **внутрисхемное программирование**, то есть прямо в схеме, не выключая питания.

При параллельном программировании данные передаются по байтам. Для передачи байта используется восемь ножек микросхемы, которые играют роль шины данных. При последовательном способе программирования для передачи данных используется всего три вывода микросхемы. Эти выводы имеют следующие названия MISO, MOSI, SCK. Разумеется, все эти выводы совмещены с выводами одного из портов. Расшифровка названий выводов следующая:

MISO — Master Input, Slave Output (Ведущее работает на ввод, ведомое — на вывод);

MOSI — Master Output, Slave Input (Ведущее работает на вывод, ведомое — на ввод);

SCK — Synchronize Clock (Сигнал синхронизации).

При последовательном программировании байты передаются побитно. Очевидно, что при параллельном способе программирования микросхема будет запрограммирована быстрее, чем при последовательном способе. Однако параллельный способ не позволяет выполнять внутрисхемное программирование.

Параллельный способ программирования имеет **две модификации:**

- параллельное программирование с повышенным питанием;
- низковольтное параллельное программирование.

Повышенное питание (+12 В) подается на вывод Reset непосредственно в момент программирования. Низковольтное программирование не требует никаких дополнительных напряжений. Питание всех цепей осуществляется от напряжения +5 В.

Последовательное программирование бывает только низковольтным. Высоковольтный режим программирования обеспечивает самую высокую скорость «прошивки» микросхем.

2.3.2. Схема программатора

Универсальные и специализированные программаторы

Как уже говорилось, в настоящее время разработано огромное множество различных схем программаторов. Их описание можно встретить в различной литературе, а также скачать из Интернета. Все схемы можно классифицировать по следующим параметрам.

Во-первых, схемы бывают универсальные и специализированные. **Универсальные схемы** позволяют программировать не один вид микросхем, а несколько видов. Причем не только в пределах одной серии, но и даже микросхемы разных серий и разных производителей. Кроме того, универсальные программаторы обычно умеют программировать также и микросхемы других типов. **Например**, ПЗУ, EEPROM и т. д.

Специализированные программаторы предназначены для программирования микросхем одной серии или даже микросхем всего одного конкретного типа.

Если говорить о микросхемах серии AVR, то программаторы могут различаться по способу программирования. Существуют программаторы, которые поддерживают оба способа (параллельный и последовательный). Упрощенные программаторы умеют программировать только последовательным способом.

Способ подключения программатора к компьютеру

Следующий параметр, по которому можно классифицировать все без исключения программаторы — **способ подключения к компьютеру**. В настоящее время существует только два способа подключения:

- при помощи параллельного порта компьютера (LPT);
- при помощи последовательного порта (COM).

Возможен и третий вариант: подключение по каналу USB. Но такие программаторы в настоящее время пока не получили широкого распространения.

Программаторы, подключаемые посредством последовательного порта, отличаются более сложной схемой по сравнению с программаторами, подключаемыми по LPT. Последовательный порт требует аналогичного порта на другом конце линии связи. Поэтому такие

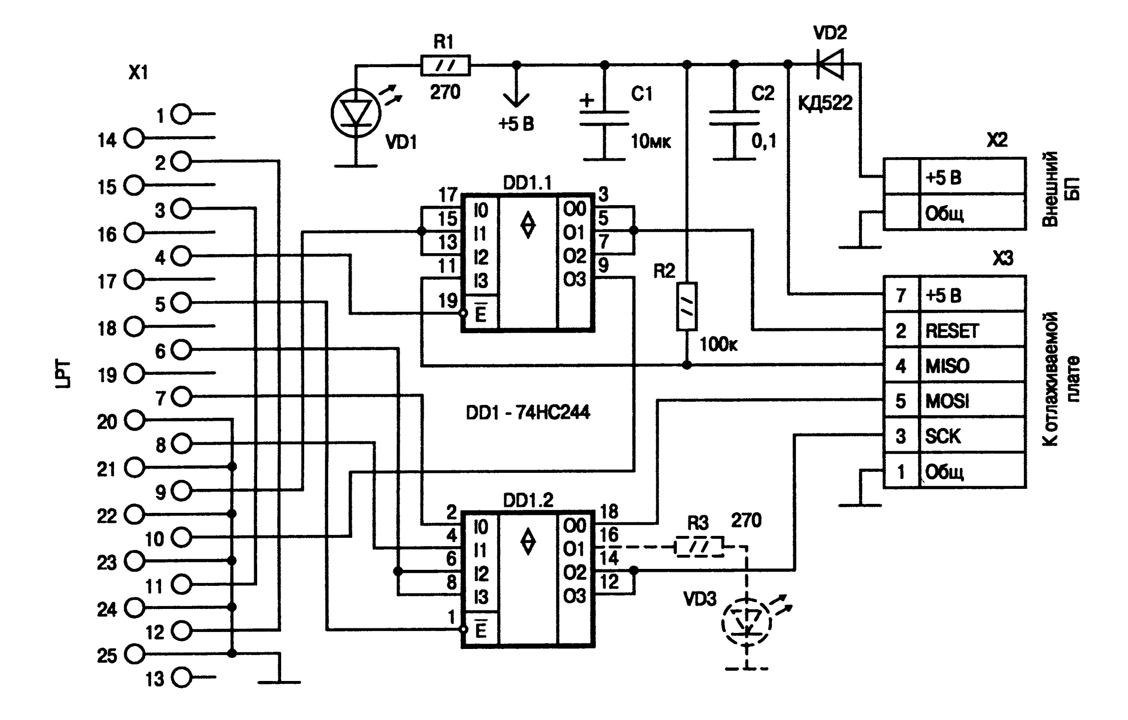


Рис. 2.8. *Схема программатора*

программаторы обычно включают в себя специальный технологический микроконтроллер и схему согласования уровней сигнала.

Программаторы, подключаемые к параллельному порту, отличаются простотой и надежностью. Простейший программатор, позволяющий программировать микросхемы AVR, содержит всего три развязывающих резистора. Весь алгоритм последовательного канала связи реализуется в компьютере программным путем.

Однако для индивидуального повторения оптимальным вариантом программатора является программатор, показанный на **рис. 2.8**. Эта схема совместима с программатором системы STK200/300.

Предлагаемый программатор предназначен только для работы в последовательном режиме и предусматривает внутрисхемное программирование. От простейшей схемы на трех резисторах данная схема отличается наличием защитного буфера на микросхеме 74HC244. Эта микросхема представляет собой два четырехканальных управляемых буфера.

Управление каждым из буферов производится при помощи входа $\bar{E}$. Сигнал логического нуля на этом входе открывает соответствующий буфер, и сигналы со входов буфера поступают на его выходы. Сигнал со входа I0 поступает на выход O0, сигнал со входа I1 на выход O1 и так далее. Если на вход $\bar{E}$ подать логическую единицу, буфер закрывается, и все его выходы переходят в высокоимпендансное состояние.

Программатор подключается к LPT-порту компьютера при помощи разъема X1. Именно через этот разъем на схему поступают управляющие сигналы от компьютера. Через разъем X2 на программатор может подаваться внешнее питание. К отлаживаемой схеме программатор подключается при помощи разъема X3.

В процессе программирования программа управления программатором включает буферы DD1.1 и DD1.2, а затем организует обмен информацией с программируемой микросхемой посредством следующих сигналов:

- **MSIO, MOSI, SCK** (в соответствии с алгоритмом передачи данных);
- **RESET** (перевод микросхемы в режим программирования и сброс).

По окончании процесса программирования оба буфера закрываются. После этого программатор не мешает работе схемы, к которой он подключен.

Внутрисхемное программирование

Для того, чтобы обеспечить возможность **внутрисхемного программирования,** необходимо при разработке схемы на микроконтроллере соблюдать следующее правило.

Правило. *На все входы, используемые для последовательного программирования (MSIO, MOSI, SCK, RESET), не должны поступать никакие мешающие сигналы. Проще всего оставить эти входы свободными.*

Если это невозможно, то старайтесь, чтобы к этим выводам были подключены только входы внешних микросхем, а не их выходы. Программные примеры в четвертой главе этой книги выполнены с учетом всех этих требований. Так, **в примерах с 1 по 9** выходы, предназначенные для последовательного программирования, оставлены свободными. **В примерах 10 и 11** с двумя из этих выводов пришлось совместить кнопку звонка и переключатель режимов работы. Поэтому для двух последних схем перед тем, как начать программирование, необходимо убедиться, что кнопка звонка отпущена, а переключатель режимов находится в положении «Работа» (контакты разомкнуты).

Питание программатора

Питание программатора может осуществляться как от внешнего источника стабилизированного напряжения +5 В через разъем X2, так и от отлаживаемой схемы через контакт 7 разъема X3. В том случае, если отлаживаемая схема потребляет не слишком большое количество энергии, возможен вариант питания отлаживаемой схемы от внешнего источника питания через разъемы X2 и X3 программатора.

Светодиод VD1 служит в качестве индикатора наличия питающего напряжения. Конденсаторы C1 и C2 — это фильтр по питанию. Диод VD2 — защитный. Он предотвращает возникновение паразитного тока при поступлении напряжения питания сразу от внешнего источника и от отлаживаемой платы. Светодиод VD3 служит для индикации режима программирования. Однако не все программы управления программатором поддерживают управление этим светодиодом.

2.3.3. Программа управления программатором

Знакомство с программой PonyProg

Приведенная выше схема может работать с любой программой, у которой имеется режим STK200/300. В частности, программная среда Code Vision AVR поддерживает этот программатор. Однако я **рекомендую** применять популярную в настоящее время **программу PonyProg**, которая позволит работать не только с Code Vision, но и с AVR Studio.

Программа PonyProg — это открытый проект. Для распространения этой программы и еще нескольких проектов в Интернете создан специальный сайт **http://www.lancos.com.** Программа также распространяется с открытой лицензией (GNU), то есть вместе с текстом программы, который разрешается изменять по своему усмотрению. Однако в пакет программы входит специальная библиотека, которая содержит текст всех основных функций, обеспечивающих процесс программирования микросхем.

На библиотеку не распространяется открытая лицензия. Ее разрешается использовать, но не разрешается изменять входящие в нее процедуры. Изменения допускаются лишь в интерфейсе программы. Такое решение делает программу более надежной в работе.

На сайте вы можете загрузить не только инсталляционный пакет самой программы, но также исполняемый файл русифицированного или украинофицированного вариантов программы. Кроме этого, там еще имеется целый набор вариантов, поддерживающий множество других языков. После инсталляции программы вы просто меняете исполняемый файл в директории программы на новый, и программа полностью русифицируется. Однако учтите, что русифицированная версия программы — это устаревшая версия. Она может не поддерживать ряд микроконтроллеров. Поэтому, если вы не нашли в списке микросхем ту, что вам необходима, проинсталлируйте программу PonyProg заново и работайте с английской версией.

При запуске программы PonyProg открывается окно заставки и раздается фирменный звук — лошадиное ржание. Если вы не желаете слушать его каждый раз при запуске, поставьте галочку в поле «Disable Sound» (выключить звук). Нажмите «Ok». Рекламная заставка закроется, и откроется основная панель программы (см. **рис. 2.9**).

Главная панель содержит всего одно основное окно, где в свою очередь могут быть открыты одно или несколько окон с разными вариантами прошивок. В верхней части главной панели традиционно располагается меню и две панели инструментов.

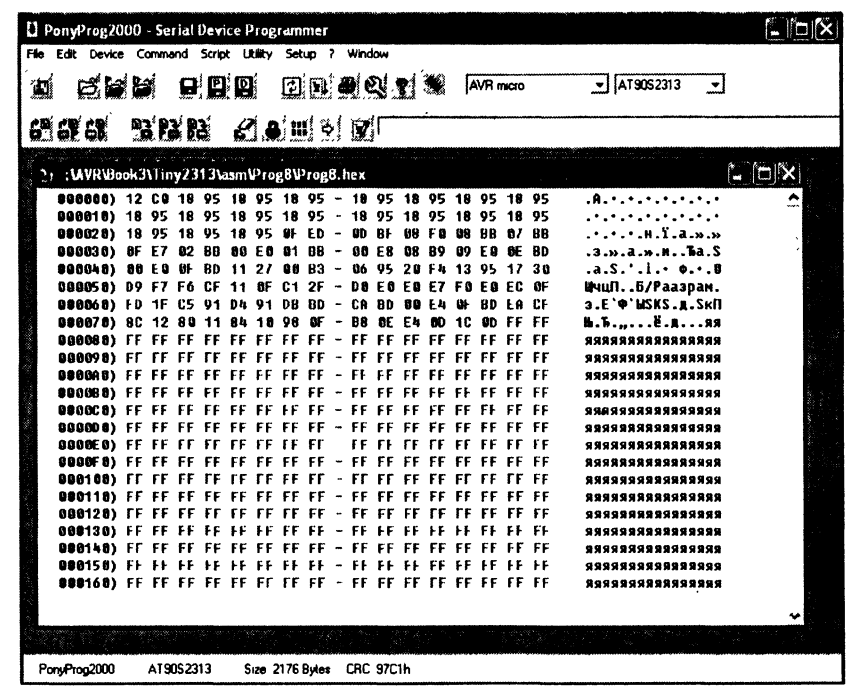


Рис. 2.9. *Основная панель программы PonyProg*

Алгоритм действий

Первое, что нужно сделать сразу после включения программы, — **выбрать тип микросхемы**, которую вы собираетесь программировать. Для этого в верхней панели инструментов имеются два выпадающих меню. Одно называется «Select device family» (Выбор семейства микросхем), а второе — «Select device type» (Выбор типа микросхем). Поля не подписаны. Названия появляются в виде всплывающей подсказки при наведении курсора мыши на соответствующее поле.

В поле выбора семейства выберите «AVR micro», а в поле выбора типа — требуемый тип микросхемы. Для всех наших примеров это будет ATiny2313. Второй способ, при помощи которого также можно выбрать семейство и тип микросхемы, — воспользоваться меню «Device» (Устройство). Выбранный тип микросхемы автоматически сохраняется, и при повторном запуске программы вызывается снова.

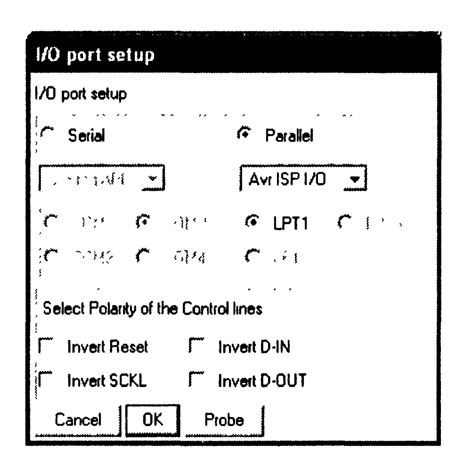


Рис. 2.10. *Окно настройки ввода—вывода*

Теперь вам необходимо **произвести настройку интерфейса** и **калибровку программы**. Эти две операции нужно выполнять только один раз, при первом запуске программы. Повторная настройка и калибровка может понадобиться лишь при сбое программы. Сначала выполним настройку интерфейса. Для этого нужно выбрать пункт «Interface Setup» меню «Setup» (см. **табл. 2.7**). Откроется окно настройки (см. **рис 2.10**).

В этом окне вы должны выбрать порт, к которому подключен ваш программатор. Кроме того, в этом окне можно проинвертировать любой из управляющих сигналов программатора, что бывает полезно при работе с нестандартными схемами. Выбираем параллельный порт (Parallel). Если ваш компьютер имеет несколько LPT-портов, выберите конкретный порт (обычно LPT1). В выпадающем списке **выберите способ программирования**. Это будет последовательное программирование по ISP-интерфейсу. Если в вашем компьютере имеется только один параллельный порт LPT1, то все установки должны выглядеть так, как это показано на **рис. 2.10**.

Таблица 2.7. *Команды меню «Setup» (Установки)*

Команда	Английский вариант	Кнопка	Описание
Настройка оборудования	Interface Setup	[wrench icon]	Открыть окно настройки интерфейса связи с компьютером
Калибровка	Calibration	-	Калибровка временных характеристик программатора

Вторая процедура, которую нужно выполнить хотя бы один раз при первом включении программы, — это **калибровка**. В процессе калибровки программа настраивает свои процедуры формирования временных интервалов под конкретный компьютер.

Цель калибровки — повышение точности формирования интервалов времени. При выполнении этой процедуры компьютер не должен выполнять никаких других программ. Закройте все открытые окна и выгрузите все программы, работающие в фоновом режиме. Затем выберите команду «Калибровка» (см. **табл. 2.7**). Появится окно предупреждения. Для старта процесса калибровки нажмите в этом окне кнопку «Ok». Процесс калибровки выполняется несколько секунд.

Программирование микросхем

После настройки и калибровки все готово для **программирования микросхем**. Для начала нам нужно загрузить данные из файлов, полученных в результате трансляции:

- файл программы для записи во Flash-память;
- файл данных для записи в EEPROM.

Для временного хранения этих данных программатор использует окно данных. Одно окно данных хранит один вариант задания (программа плюс данные). Запущенная программа Pony Prog обязательно содержит хотя бы одно такое окно. Пустое окно автоматически создается при запуске программы. После загрузки информации (программы или данных) в окне появляется дамп памяти.

Определение. *Дамп — это широко распространенный способ представления цифровых данных. Он представляет*

собой таблицу шестнадцатиричных чисел, записанных рядами по 16 чисел в ряду (см. ***рис. 2.9****).*

В начале каждого ряда записывается адрес первой его ячейки. Затем, правее, эти же шестнадцать чисел повторяются в символьном виде. То есть вместо каждого числа записывается соответствующий ему символ в кодировке ASCII.

В окно помещается сначала содержимое программной памяти микроконтроллера, а затем содержимое EEPROM. На **рис. 2.9** показан программатор с загруженной программой из примера номер 8. Из рисунка видно, что программа занимает первые восемь строчек. Причем занимает неполностью. С адреса 0x000000 по адрес 0x00007D. Дальше программная память пуста. Для того, чтобы зря не прошивать пустые ячейки, в них записан код 0xFF. Так как выбранная нами микросхема имеет объем программной памяти, равный 2 Кбайт, дамп программной памяти оканчивается ячейкой с адресом 0x0007FF. Но на этом дамп не заканчивается.

За содержимым программной памяти следует содержимое EEPROM. Причем адресация продолжается так, как будто содержимое обоих видов памяти составляет единое адресное пространство. Ячейка EEPROM с адресом 0x0000 в этой новой системе координат получает адрес 0x000800. И так далее, по нарастающей. Содержимое EEPROM на экране выделяется цветом. Пролистав дамп при помощи полосы прокрутки, вы сами можете в этом убедиться.

Размещение всей информации в едином адресном пространстве удобно, так как позволяет хранить программу и данные в одном файле. В процессе программирования микросхемы программатор автоматически отделяет программу от данных, используя информацию об объеме программной памяти данного конкретного микроконтроллера. Все, что выше этого объема автоматически считается данными для EEPROM.

Для загрузки данных из файла, находящегося на жестком диске, в текущее окно программатора, а также для записи информации из окна программатора в файл программа поддерживает ряд команд, объединенных в меню «File». Все эти и другие команды меню «File» приведены в **табл. 2.8**. В таблице приведены названия пунктов меню как на русском, так и на английском языках. А также показан внешний вид соответствующей этому пункту кнопки на панели инструментов.

Таблаца 2.8. *Команды меню «File» (Файл)*

Команда	Английский вариант	Кнопка	Описание
Новое окно	New Window		Открыть новое пустое окно
Открыть файл с данными	Open Device file		Открыть полный файл с информацией для прошивки (программа и данные) и поместить ее в текущее окно
Открыть файл программы (Flash)	Open Program (Flash) File		Открыть файл с программой, предназначенной для прошивки, и поместить ее в текущее окно в область программы
Открыть файл данных (EEPROM)	Open Data (EEPROM) File		Открыть файл с данными для прошивки в EEPROM и поместить их в текущее окно в область данных
Сохранить файл с данными	Save Device File		Сохранить данные из текущего окна в файл (полный файл данных)
Сохранить файл с данными как —	Save Device File As—	-	Сохранить данные из текущего окна в виде нового файла с другим именем
Сохранить файл программ (Flash) как—	Save Program (Flash) File As—		Сохранить информацию из области программы текущего окна в файл (Информация для Flash)
Сохранить файл данных (EEPROM) как —	Save Data (EEPROM) File As—		Сохранить информацию из области данных текущего окна в файл (информация для EEPROM)
Открыть заново	Reload Files		Заново перечитать информацию текущего окна из файла
Печать	Print—		Открыть окно вывода на печать содержимого текущего окна
Закрыть	Close	-	Закрыть текущее окно. Если это последнее окно, то закрывается вся программа
Выход	Exit	-	Завершение работы программы

Итак, **загрузим программу и данные в программатор**. Если вы помните, все вышеперечисленные трансляторы создают отдельный файл для программы (файл с расширением hex) и отдельный файл для данных (файл с расширением еер). Поэтому для загрузки программы воспользуемся командой «Открыть файл программы (Flash)». При выборе этой команды появляется диалог «Открыть программу». Убедитесь, что в поле «Тип файла» выбрано «* . hex». Если это не так, выберите это значение из выпадающего списка.

Затем найдите на диске директорию вашего проекта, выберите файл и нажмите кнопку «Открыть». Загруженные данные появятся в текущем окне. Таким же образом загружается содержимое EEPROM. Только в этом случае нужно выбрать тип файла «*.eep».

После того, как программа и данные загружены, их можно просмотреть, при необходимости — подредактировать прямо в окне программатора. А если нужно, то и записать обратно на диск. Если у вас есть принтер, можно распечатать дамп из текущего окна на бумаге.

Но основная функция — это, естественно, **запись программы и данных в память микроконтроллера**. Все команды, предназначенные для работы с микроконтроллером, сведены в меню «Command». Их описание приведено в **табл. 2.9**. При помощи этих команд вы можете отдельно запрограммировать память программ, отдельно — EEPROM. Команда «Записать все» позволяет запрограммировать программу и данные за одну операцию.

Три команды считывания позволяют прочитать содержимое памяти программ и памяти данных микроконтроллера. Прочитанные данные помещаются в текущее окно программатора. Считанную из микросхемы информацию можно записать на диск при помощи команд, описанных в **табл. 2.8**. Группа команд проверки используется для сравнения информации, записанной в микросхему, и информации в текущем окне программатора.

Команда «Стереть» позволяет стереть память микросхемы. Команда стирает одновременно все виды памяти:

- память программ;
- память данных
- ячейки защиты (если они были запрограммированы).

Однако здесь есть одно исключение. Некоторые микросхемы (в том числе и ATiny2313) имеют бит конфигурации (fuse-переключатель), запрещающий стирание EEPROM. Если запрограммировать этот бит, то при стирании микросхемы EEPROM стираться не будет. Это позволяет не делать лишних циклов записи/стирания и сэкономить ресурс EEPROM в том случае, когда его содержимое менять не обязательно.

На **пункте меню «Биты защиты и конфигурации»** необходимо остановиться подробнее. Эта команда предназначена для чтения и изменения fuse-переключателей (битов конфигурации) и битов защиты микросхемы. В русскоязычном варианте программы этот пункт почему-то остался не переведенным.

Таблаца 2.9. *Команды меню «Command» (Команды)*

Команда	Английский вариант	Кнопка	Описание
Считать все	Read All		Прочитать программу (из Flash) и данные (из EEPROM) из микросхемы и поместить в текущее окно
Считать программу (Flash)	Read Program (FLASH)		Прочитать программу (из Flash) из микросхемы и поместить в текущее окно в область программы
Считать данные (EEPROM)	Read DATA (EEPROM)		Прочитать данные (из EEPROM) из микросхемы и поместить в текущее окно в область данных
Записать все	Write All		Записать программу и данные из текущего окна в микросхему
Записать программу (Flash)	Write Program (FLASH)		Записать программу из текущего окна во Flash-память микросхемы
Записать данные (EEPROM)	Write DATA (EEPROM)		Записать данные из текущего окна в EEPROM-память микросхемы
Проверить все	Verify All	-	Сравнить программу и данные из текущего окна с содержимым соответственно Flash- и EEPROM-памяти микросхемы
Проверить программу (Flash)	Verify Program (FLASH)	-	Сравнить информацию из области программ текущего окна с информацией во Flash-памяти микросхемы
Проверить данные (EEPROM)	Verify DATA (EEPROM)	-	Сравнить информацию из области данных текущего окна с информацией в EEPROM-памяти микросхемы
Биты защиты и конфигурации	Security and Configuration Bits—		Открыть окно чтения/изменения битов защиты и битов конфигурации микросхемы
Стереть	Erase		Стереть Flash- и EEPROM-память микросхемы
Аппаратный сброс	Reset	-	Подать на микросхему сигнал аппаратного сброса
Программирование	Program		Выполнение последовательности команд, составляющих цикл программирования
Настройка программирования	Program Options—	-	Настройка цикла программирования (выбор команд)
Считать калибровочный бит ген.	Read Osc. Calibration Byte	-	Чтение значения калибровочного бита для дальнейшего использования в целях калибровки генератора
Настройка калибровки генератора	Osc. Calibration Options—	-	Настройка процедуры автоматической записи калибровочного значения в память данных или программ

При выборе этого пункта меню открывается окно, показанное на **рис. 2.11**. Набор элементов управления для каждого вида микросхем будет свой. Причем сразу после открытия окна все поля не будут

выбраны (не будут содержать «галочек»). Это значит, что содержимое этих полей пока не соответствует реальному содержимому битов защиты и конфигурации микросхемы.

Для того, чтобы считать эти значения, необходимо нажать в том же окне кнопку «Read». На короткий момент появится окно, показывающее процесс считывания. Затем снова откроется окно битов защиты и конфигурации. Теперь уже все поля примут значения, считанные из микросхемы. Галочка в любом из полей означает, что данный бит запрограммирован. **Напоминаю**, что запрограммированный бит содержит ноль, незапрограммированный — единицу. Теперь вы можете изменить значение любого бита. Но эти изменения будут только на экране. Для того, чтобы записать изменения в микросхему, нажмите кнопку «Write». Кнопки «Set All» и «Clear All» позволяют установить или сбросить значения сразу всех полей в данном окне.

Для всех примеров, приведенных в **главе 1**, необходимо установить такое значение битов, как показано на **рис. 2.11**. Если вы хотите обойтись без кварцевого резонатора, то достаточно поменять значения битов CKSEL3—CKSEL0 следующим образом:

CKSEL3 — 0 (есть галочка);
CKSEL2 — 0 (есть галочка);
CKSEL1 — 1 (нет галочки);
CKSEL0 — 0 (есть галочка).

Значения всех остальных битов нужно оставить без изменений. Когда вы запишите в микроконтроллер это измененное значение fuse-битов, то при использовании такой микросхемы в любом из наших программных примеров из схемы можно исключить кварцевый резонатор Q1 и конденсаторы C1 и C2.

Выводы XTAL0 и XTAL1 при этом останутся незадействованными. Все измененные таким образом схемы прекрасно работают. Единственный **недостаток внутреннего генератора** — низкая стабильность частоты. Однако ни один из наших примеров не критичен к стабильности частоты. Разве что тональность мелодий будет немного изменяться. Но на слух это обнаружить вряд ли удастся.

Две команды меню «Command» предназначены для калибровки внутреннего тактового генератора микроконтроллера. В микросхеме ATiny2313 эта операция выполняется автоматически. Однако есть ряд микросхем, в которых значение калибровочного байта нужно вручную внести в одну из ячеек программной памяти

процессора (или EEPROM). В программаторе эта операция автоматизирована.

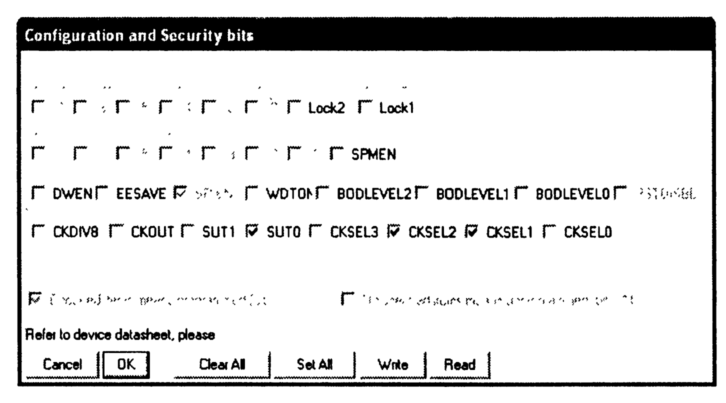


Рис. 2.11. *Окно установки битов конфигурации и защиты*

Сначала вы должны определиться с адресом ячейки. Вы можете выбрать любую заведомо свободную ячейку в памяти программ или в EEPROM. Ваша программа должна быть составлена таким образом, чтобы в начале своей работы она читала содержимое этой ячейки и записывала его в регистр калибровки тактового генератора (OSCCAL). Когда адрес ячейки известен, нужно настроить систему автоматического считывания байта калибровки в программаторе.

Для этого сначала выберите команду «Настройка калибровки генератора» (см. **табл. 2.9**). Откроется небольшое окно, куда вы должны ввести адрес ячейки. В том же окне имеется поле «Data memory offset» (Относительно памяти данных). Если поставить галочку в этом поле, то калибровочный байт будет записываться в EEPROM. Выбрав таким образом адрес и месторасположение ячейки, нажимаем кнопку «OK».

Теперь перед каждым программированием вам достаточно подать команду «Считать калибровочный байт ген.», и калибровочный байт будет прочитан и помещен по указанному вами адресу в текущее окно программатора. При записи информации в микроконтроллер калибровочное значение попадет в предназначенную ему ячейку автоматически.

Режимы работы программатора

Для удобства работы с программатором он имеет **режим группового выполнения команд**. Команды чтения информации из файла, чтения байта конфигурации, обновления серийного номера, стирания микросхемы и, наконец, программирования могут выполняться в пакете при нажатии всего одной кнопки. Для настройки пакета команд выберите пункт «Настройка программирования» меню «Command». Откроется окно «Program Options» (см. **рис. 2.12**). В этом окне отметьте галочками те операции, которые должны выполняться при запуске пакета, и нажмите кнопку «OK». Для запуска пакетной команды достаточно выбрать пункт «Программирование» (см. **табл. 2.9**). Или нажать кнопку [⇨] на панели инструментов.

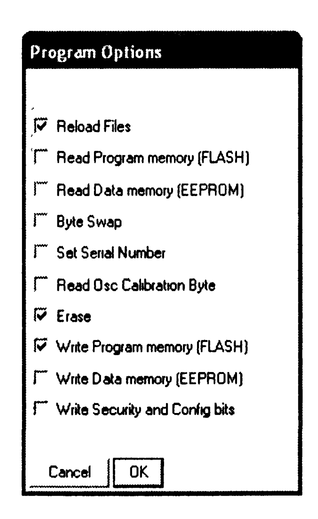


Рис. 2.12. *Настройка пакетного выполнения команд*

Пакетный режим очень удобен в процессе отладки программы. Если в пакет включена опция «Reload Files», то достаточно один раз вручную открыть нужный файл нужного проекта, а затем можно просто нажимать кнопку «Программирование» каждый раз, когда нужно перепрошить микросхему новой версией программы. Все остальное PoniProg сделает за вас. Новая версия программы сама загрузится в программатор, микросхема сотрется, а затем в нее запишется новая информация.

Только не забывайте поставить команду стирания, если вы собираетесь программировать. Помните, что при записи в нестертую микросхему результат непредсказуем. Каждый «прошитый» в процессе программирования бит может быть восстановлен только в результате стирания всей памяти.

И, в заключение, хочу рассказать об еще одной удобной функции программатора. Программатор имеет **встроенную систему автоматического формирования серийного номера программы**.

Серийный номер — это просто порядковый номер версии программы. Этот номер может автоматически записываться в выбранную вами ячейку памяти программ или памяти данных. Настройка данного режима производится при выборе пункта «SerialNumber Config...» (Установки серийного номера) меню «Utility» (Утилиты).

В открывшемся окне вы можете выбрать адрес ячейки для серийного номера, поставить галочку в поле «Data memory offset» (Относительно памяти данных), а также выбрать параметры его автоматического изменения. После настройки параметров изменение серийного номера и его запись в выбранную ячейку текущего окна программатора производится путем выбора пункта «Set Serial Number» (Установить серийный номер) меню «Utility» (Утилиты) или нажатием кнопки .

ПРИЛОЖЕНИЕ

В приложении к данной книге в качестве бонуса прилагается ***сводная таблица команд Ассемблера микроконтроллеров AVR***

- *группа команд логических операций*
- *группа команд арифметических операций*
- *группа команд операций с разрядами*
- *группа команд сравнения*
- *группа команд операций сдвига*
- *группа команд пересылки данных*
- *группа команд управления системой*
- *группа команд передачи управления (безусловная передача управления)*
- *группа команд передачи управления (пропуск команды по условию)*
- *группа команд передачи управления (передача управления по условию)*

Сводная таблица команд Ассемблера микроконтроллеров AVR

Группа команд логических операций

Мнемоника	Описание	Операция	Циклы	Флаги
AND Rd, Rr	«Логическое И» двух РОН	Rd ← Rd · Rr	1	Z, N, V
ANDI Rd, K	«Логическое И» РОН и константы	Rd ← Rd · K	1	Z, N, V
EOR Rd, Rr	«Исключающее ИЛИ» двух РОН	Rd ← Rd ⊕ Rr	1	Z, N, V
OR Rd, Rr	«Логическое ИЛИ» двух РОН	Rd ← Rd v Rr	1	Z, N, V
ORI Rd, K	«Логическое ИЛИ» РОН и константы	Rd ← Rd v K	1	Z, N, V
COM Rd	Перевод в обратный код	Rd ← 0FFH — Rd	1	Z, C, N, V
NEG Rd	Перевод в дополнительный код	Rd ← 00H — Rd	1	Z, C, N, V, H
CLR Rd	Сброс всех разрядов РОН	Rd ← Rd ⊕ Rd	1	Z, N, V
SER Rd	Установка всех разрядов РОН	Rd ← 0FFH	1	—
TST Rd	Проверка РОН на отрицательное (нулевое) значение	Rd ← Rd · Rd	1	Z, N, V

Группа команд арифметических операций

Мнемоника	Описание	Операция	Циклы	Флаги
ADD Rd, Rr	Сложение двух РОН	Rd ← Rd + Rr	1	Z, C, N, V, H
ADC Rd, Rr	Сложение двух РОН с переносом	Rd ← Rd + Rr + C	1	Z, C, N, V, H
ADIW Rd, K	Сложение регистровой пары с константой	Rdh:Rdl ← Rdh: Rdl + K	2	Z, C, N, V, S
SUB Rd, Rr	Вычитание двух РОН	Rd ← Rd — Rr	1	Z, C, N, V, H
SUBI Rd, K	Вычитание константы из РОН	Rd ← Rd — K	1	Z, C, N, V, H
SBC Rd, Rr	Вычитание двух РОН с заемом	Rd ← Rd — Rr — C	1	Z, C, N, V, H
SBCI Rd, K	Вычитание константы из РОН с заемом	Rd ← Rd — K — C	1	Z, C, N, V, H
SBIW Rd, K	Вычитание константы из регистровой пары	Rdh:Rdl ← Rdh: Rdl — K	2	Z, C, N, V, S
DEC Rd	Декремент РОН	Rd ← Rd — 1	1	Z, N, V
INC Rd	Инкремент РОН	Rd ← Rd + 1	1	Z, N, V

Группа команд операций с разрядами

Мнемоника	Описание	Операция	Циклы	Флаги
CBR Rd, K	Сброс разряда(ов) РОН	Rd ← Rd • (0FFH — K)	1	Z,N,V
SBR Rd, K	Установка разряда(ов) РОН	Rd ← Rd K	1	Z,N,V
CBI A, b	Сброс разряда РВВ	A.b ← 0	2	—
SBI A, b	Установка разряда РВВ	A.b ← 1	2	—
BCLR s	Сброс флага	SREG.s ← 0	1	SREG.s
BSET s	Установка флага	SREG.s ← 1	1	SREG.s
BLD Rd, b	Загрузка разряда РОН из флага T (SREG)	Rd.b ← T	1	—
BST Rr, b	Запись разряда РОН в флаг T (SREG)	T ← Rd.b	1	T
CLC	Сброс флага переноса	C ← 0	1	C
SEC	Установка флага переноса	C ← 1	1	C
CLN	Сброс флага отрицательного числа	N ← 0	1	N
SEN	Установка флага отрицательного числа	N ← 1	1	N
CLZ	Сброс флага нуля	Z ← 0	1	Z
SEZ	Установка флага нуля	Z ← 1	1	Z
CLI	Общий запрет прерываний	I ← 0	1	I
SEI	Общее разрешение прерываний	I ← 1	1	I
CLS	Сброс флага знака	S ← 0	1	S
SES	Установка флага знака	S ← 1	1	S
CLV	Сброс флага переполнения дополнительного кода	V ← 0	1	V
SEV	Установка флага переполнения дополнительного кода	V ← 1	1	V
CLT	Сброс пользовательского флага T	T ← 0	1	T
SET	Установка пользовательского флага T	T ← 1	1	T
CLH	Сброс флага половинного переноса	H ← 0	1	H
SEH	Установка флага половинного переноса	H ← 1	1	H

Группа команд сравнения

Мнемоника	Описание	Операция	Циклы	Флаги
CP Rd, Rr	Сравнение двух РОН	? (Rd — Rr)	1	Z,N,V,C,H
CPC Rd, Rr	Сравнение РОН с учетом переноса	? (Rd — Rr — C)	1	Z,N,V,C,H
CPI Rd, K	Сравнение РОН с константой	? (Rd — K)	1	Z,N,V,C,H

Группа команд операций сдвига

Мнемоника	Описание	Операция	Циклы	Флаги
ASR Rd	Арифметический сдвиг вправо	Rd7 → Rd6 → Rd5 → Rd4 → Rd3 → Rd2 → Rd1 → Rd0	1	Z,C,N,V
LSL Rd	Логический сдвиг влево	C ← Rd7 ← Rd6 ← Rd5 ← Rd4 ← Rd3 ← Rd2 ← Rd1 ← Rd0 ← 0	1	Z,C,N,V
LSR Rd	Логический сдвиг вправо	0 → Rd7 → Rd6 → Rd5 → Rd4 → Rd3 → Rd2 → Rd1 → Rd0 → C	1	Z,C,N,V
ROL Rd	Сдвиг влево через перенос	C ← Rd7 ← Rd6 ← Rd5 ← Rd4 ← Rd3 ← Rd2 ← Rd1 ← Rd0 ← C	1	Z,C,N,V
ROR Rd	Сдвиг вправо через перенос	C → Rd7 → Rd6 → Rd5 → Rd4 → Rd3 → Rd2 → Rd1 → Rd0 → C	1	Z,C,N,V
SWAP Rd	Обмен местами тетрад	Rd(3—0) ↔ Rd(7—4)	1	—

Группа команд пересылки данных

Мнемоника	Описание	Операция	Циклы	Флаги
MOV Rd, Rr	Пересылка между РОН	Rd ← Rr	1	—
MOVW Rd, Rr	Пересылка между парами регистров	Rd +1:Rd ← Rr+1:Rr	1	—
LDI Rd, K	Загрузка константы в РОН	Rd ← K	1	—
LD Rd, X	Косвенное чтение	Rd ← [X]	2	—
LD Rd, X+	Косвенное чтение с постикрементом	Rd ← [X], X ← X+1	2	—
LD Rd, -X	Косвенное чтение с преддекрементом	X ← X-1, Rd ← [X]	2	—
LD Rd, Y	Косвенное чтение	Rd ← [Y]	2	—
LD Rd, Y+	Косвенное чтение с постикрементом	Rd ← [Y], Y ← Y+1	2	—
LD Rd, –Y	Косвенное чтение с преддекрементом	Y ← Y-1, Rd ← [Y]	2	—
LD Rd, Y+q	Косвенное относительное чтение	Rd ← [Y+q]	2	—
LD Rd, Z	Косвенное чтение	Rd ← [Z]	2	—
LD Rd, Z+	Косвенное чтение с постикрементом	Rd ← [Z], Z ← Z+1	2	—
LD Rd, -Z	Косвенное чтение с преддекрементом	Z ← Z-1, Rd ← [Z]	2	—
LD Rd, Z+q	Косвенное относительное чтение	Rd ← [Z+q]	2	—
LDS Rd, k	Непосредственное чтение из ОЗУ	Rd ← [k]	2	—
ST X, Rr	Косвенная запись	[X] ← Rr	2	—
ST X+, Rr	Косвенная запись с постикрементом	[X] ← Rr, X ← X+1	2	—
ST -X, Rr	Косвенная запись с преддекрементом	X ← X-1, [X] ← Rr	2	—
ST Y, Rr	Косвенная запись	[Y] ← Rr	2	—
ST Y+, Rr	Косвенная запись с постикрементом	[Y] ← Rr, Y ← Y+1	2	—
ST -Y, Rr	Косвенная запись с преддекрементом	Y ← Y-1, [Y] ← Rr	2	—
ST Y+q, Rr	Косвенная относительная запись	[Y+q] ← Rr	2	—
ST Z, Rr	Косвенная запись	[Z] ← Rr	2	—
ST Z+, Rr	Косвенная запись с постикрементом	[Z] ← Rr, Z ← Z+1	2	—

Мнемоника	Описание	Операция	Циклы	Флаги
ST -Z, Rr	Косвенная запись с преддекрементом	Z ← Z-1, [Z] ← Rr	2	—
ST Z+q, Rr	Косвенная относительная запись	[Z+q] ← Rr	2	—
STS k, Rr	Непосредственная запись в ОЗУ	[k] ← Rr	2	—
LPM	Загрузка данных из памяти программ	R0 ← {Z}	3	—
LPM Rd, Z	Загрузка данных из памяти программ	Rd ← {Z}	3	—
LPM Rd, Z+	Загрузка данных из памяти программ и постдекремент Z	Rd ← {Z}, Z ← Z+1	3	—
SPM	Запись в программную память	{Z} ← R1:R0	-	—
IN Rd, P	Пересылка из РВВ в РОН	Rd ← P	1	—
OUT P, Rr	Пересылка из РОН в РВВ	P ← Rr	1	—
PUSH Rr	Сохранение байта в стеке	STACK ← Rr	2	—
POP Rd	Извлечение байта из стека	Rd ← STACK	2	—

Группа команд управления системой

Мнемоника	Описание	Операция	Циклы	Флаги
NOP	Нет операции	-	1	—
SLEEP	Переход в «спящий» режим	-	3	—
WDR	Сброс сторожевого таймера	-	1	—
BREAK	Приостановка программы	Используется только при отладке	-	—

Группа команд передачи управления (безусловная передача управления)

Мнемоника	Описание	Операция	Циклы	Флаги
RJMP	Относительный безусловный переход	PC ← PC + k + 1	2	—
IJMP	Косвенный безусловный переход	PC ← Z	2	—
RCALL	Относительный вызов подпрограммы	PC ← PC + k + 1	3	—
ICALL	Косвенный вызов подпрограммы	PC ← Z	3	—
RET	Возврат из подпрограммы	PC ← STACK	4	—
RETI	Возврат из подпрограммы обработки прерываний	PC ← STACK	4	I

Группа команд передачи управления (пропуск команды по условию)

Мнемоника	Описание	Условие	Циклы	Флаги
Все команды этой группы пропускают следующую за ней команду (PC ← PC + 1) при разных условиях:				
CPSE Rd, Rr	Сравнение и пропуск следующей команды при равенстве	Если Rd = Rr	1/2/3	—
SBRC Rr, b	Пропуск следующей команды если разряд РОН сброшен	Если Rr,b = 0	1/2/3	—
SBRS Rr, b	Пропуск следующей команды если разряд РОН установлен	Если Rr.b = 1	1/2/3	—
SBIC A, b	Пропуск следующей команды если разряд РВВ сброшен	Если A.b = 0	1/2/3	—
SBIS A, b	Пропуск следующей команды если разряд РВВ установлен	Если A.b = 1	1/2/3	—

Группа команд передачи управления (передача управления по условию)

Мнемоника	Описание	Условие	Циклы	Флаги
Все команды этой группы выполняют переход (PC ← PC + k + 1) при разных условиях:				
BRBC s, k	Переход, если флаг s регистра SREG сброшен	Если SREG.s = 0	1/2	—
BRBS s, k	Переход, если флаг s регистра SREG установлен	Если SREG.s = 1	1/2	—
BRCS k	Переход по переносу	Если C = 1	1/2	—
BRCC k	Переход, если нет переноса	Если C = 0	1/2	—
BREQ k	Переход по условию «равно»	Если Z = 1	1/2	—
BRNE k	Переход по условию «неравно»	Если Z = 0	1/2	—
BRSH k	Переход по условию «больше или равно»	Если C = 0	1/2	—
BRLO k	Переход по условию «меньше»	Если C = 1	1/2	—
BRMI k	Переход по условию «отрицательное значение»	Если N = 1	1/2	—
BRPL k	Переход по условию «положительное значение»	Если N = 0	1/2	—
BRGE k	Переход по условию «больше или равно» (со знаком)	Если (N V) = 0	1/2	—
BRLT k	Переход по условию «меньше» (со знаком)	Если (N V) = 1	1/2	—
BRHS k	Переход по половинному переносу	Если H = 1	1/2	—
BRHC k	Переход, если нет половинного переноса	Если H = 0	1/2	—
BRTS k	Переход, если флаг T установлен	Если T = 1	1/2	—
BRTC k	Переход, если флаг T сброшен	Если T = 0	1/2	—
BRVS k	Переход по переполнению дополнительного кода	Если V = 1	1/2	—
BRVC k	Переход, если нет переполнения дополнительного кода	Если V = 0	1/2	—
BRID k	Переход, если прерывания запрещены	Если I = 0	1/2	—
BRIE k	Переход, если прерывания разрешены	Если I = 1	1/2	—

Список литературы

1. *Евстифеев А. В.* Микроконтроллеры AVR семейства Tiny и Mega фирмы «Atmel». — М.: Издательский дом «Додэка». — 2004
2. *Белов А. В.* Конструирование устройств на микроконтроллерах. — Санкт-Петербург: Наука и Техника. — 2005
3. *Белов А.В.* Самоучитель по микропроцессорной технике. Изд. 2-е, перераб. и доп. — Санкт-Петербург: Наука и Техника. — 2007
4. *Белов А.В.* Микроконтроллеры AVR в радиолюбительской практике. — Санкт-Петербург: Наука и Техника. — 2007
5. Выбор коэффициентов деления частоты. — Радио, №3. — 1990. — С. 63-64.

Список ссылок в Интернет

http://book.microprocessor.by.ru — официальный сайт книги

http://avbelov.by.ru — сайт автора

http://www.atmel.com — сайт фирмы Atmel, производителя микроконтроллеров AVR

http://www.atmel.ru — русскоязычная версия сайта Atmel

http://www.atmel.ru/Software/Software.htm — адрес для скачивания программы AVR Studio

http://www.hpinfotech.ro — сайт разработчика программы CodeVisionAVR

http://www.lancos.com — сайт программы Pony Prog.

http://www.nit.com.ru — официальный сайт издательства Наука и Техника и интернет-магазин с возможностью отправки книг издательства в любую страну мира.

Закажите лучшие книги для радиолюбителей

500 схем для радиолюбителей



2-е издание, перераб. и доп.

ISBN: 5-94387-244-2
Формат: 140×205 мм
Объем: 416 с.
Цена: 142 руб.

Заказ____экз.

2-е издание, перераб. и доп.

ISBN: 5-94387-168-3
Формат: 140×205 мм
Объем: 272 с.
Цена: 109 руб.

Заказ____экз.



ISBN: 5-94387-185-3
Формат: 140×205 мм
Объем: 272 с.
Цена: 120 руб.

Заказ____экз.

Семьян А.П.

Названия этих книг начинаются словами «500 схем...», с уточняющими названиями «Приемники», «Источники питания», «Радиостанции и трансиверы» и др. В этих книгах собраны наиболее интересные схемы полезных устройств, дается возможность каждому радиолюбителю выбрать то, что ему необходимо из великого множества схем и конструкций, проверенных и испытанных на практике.

Звуковая схемотехника для радиолюбителей



ISBN: 5-94387-082-2
Формат: 140×205 мм
Объем: 400 с.
Цена: 120 руб.

Заказ____экз.

Петров А.А.

В книге изложен обширный материал по вопросам, касающимся акустики, грамзаписи, магнитной звукозаписи, конструирования различных узлов звуковой аппаратуры. Приведены наиболее интересные схемотехнические решения, практические схемы различных устройств. Рассмотрены вопросы проектирования устройств электропитания бытовой РЭА (расчет трансформаторов, дросселей, стабилизаторов напряжения, теплоотводов).

Радиостанция своими руками



ISBN: 5-94387-085-7
Формат: 140×205 мм
Объем: 144 с.
Цена: 87 руб.

Заказ____экз.

Шмырев А.А.

Книга поможет радиолюбителю при минимальных затратах создать приемо-передающий комплекс с хорошими характеристиками. Материал изложен подробно, с полными электрическими данными, с рисунками печатных плат, с методикой настройки трансиверной приставки. Книга содержит информацию о том, как с помощью изготовленной приставки работать с цифровыми видами связи, подключить к приставке ПК.

На вклейке представлены принципиальные схемы радиоприемников Р-250М и Р-250М2.

Принимаются ксерокопии.

Закажите лучшие книги для радиолюбителей

Энциклопедия радиолюбителя. Работаем с компьютером



ISBN: 5-94387-117-9
Формат: 165×235 мм
Объем: 272 с.
Цена: 87 руб.

Заказ____экз.

Пестриков В.М.

Энциклопедия радиолюбителя содержит системные научно-популярные сведения по различным вопросам практической радиоэлектроники: конструированию и изготовлению устройств, использованию компьютера в радиолюбительской практике. Основная цель книги — помочь всем интересующимся радиоэлектроникой разобраться в ее азах и сделать в ней первые практические шаги.

Радиоэлектроника в конструкциях и увлечениях



ISBN: 5-94387-124-1
Формат: 165×235 мм
Объем: 240 с.
Цена: 87 руб.

Заказ____экз.

Пестриков В.М.

В книге представлены различные области любительской радиоэлектроники, которые могут вызвать интерес не только у тех, кто умеет держать в руках паяльник, но и у тех, кто не умеет этого делать, но желает с интересом провести свой досуг. Уделено внимание использованию компьютера в радиолюбительской практике и работе в Интернете.

Для любителей электронной музыки даны материалы по конструированию электрогитар и устройств.

Ламповый Hi-Fi усилитель своими руками



2-е издание, перераб. и доп.

ISBN: 5-94387-177-2
формат: 140×205 мм
Объем: 272 с.
Цена: 131 руб.

Заказ____экз.

Торопкин М.В.

Книга адресована любителям высококачественного звуковоспроизведения. Следование приведенному материалу позволит собрать свой первый Hi-Fi ламповый усилитель. Для начинающих радиолюбителей представлена глава «Основы схемотехники ламповых усилительных каскадов». Тем, кто решил приобрести готовый усилитель или сравнить характеристики моделей заводского изготовления будет интересна глава «Обзор рынка ламповых Hi-Fi усилителей.

Аудиосистема класса Hi-Fi своими руками: советы и секреты



ISBN: 5-94387-226-4
Формат: 140×205 мм
Объем: 272 с.
Цена: 131 руб.

Заказ____экз.

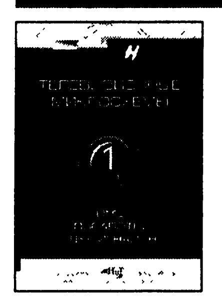
Андреев Д. А.
Торопкин М. В.

Данная книга является находкой для желающих обзавестись высококачественной аудиосистемой. Задача авторов — помочь всем категориям читателей в выборе, а имеющим радиолюбительскую подготовку — в изготовлении компонентов аудиосистемы.

Большое значение имеют приводимые в книге технологии и советы категории «ноу-хау», без обладания которыми можно лишь повторить конструкции, но нельзя добиться подлинно высококачественного звучания.

Принимаются ксерокопии.

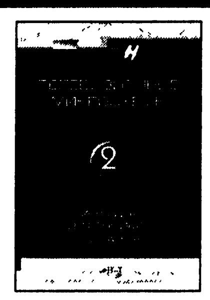
Современный четырехтомник «Телевизионные микросхемы»



Том 1. ИМС обработки телевизионных сигналов

ISBN: 5-94387-143-8
Формат: 165×235 мм
Объем: 288 с.
Цена: 142 руб.

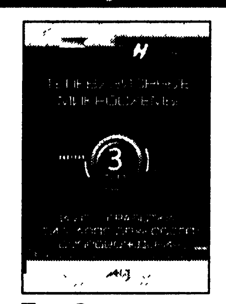
Заказ____экз.



Том 2. ИМС для источников питания

ISBN: 5-94387-147-0
Формат: 165×235 мм
Объем: 192 с.
Цена: 142 руб.

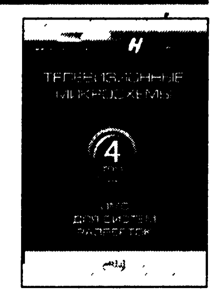
Заказ____экз.



Том 3. ИМС обработки сигналов звукового сопровождения

ISBN: 5-94387-157-8
Формат: 165×235 мм
Объем: 240 с.
Цена: 142 руб.

Заказ____экз.



Том 4. ИМС для систем развертки

ISBN: 5-94387-148-9
Формат: 165×235 мм
Объем: 208 с.
Цена: 142 руб.

Заказ____экз.

Для рассмотрения выбраны наиболее популярные ТВ микросхемы. Они сгруппированы по томам в соответствии с функциональным назначением. В каждом томе ИМС расположены в алфавитном порядке. Для микросхем приводятся: основные функции, цоколевка, назначение выводов, структурная схема, типовая схема включения. Имеется алфавитный указатель ИМС.

Видеопроцессоры. Справочник



ISBN: 5-94387-081-4
Формат: 165×235 мм
Объем: 256 с.
Цена: 142 руб.

Заказ____экз.

В справочнике основное внимание уделено современным видеопроцессорам. Приводятся структурные схемы и типовые схемы включения микросхем. Описаны схемотехнические решения и принципы работы всех функциональных устройств, вошедших в состав видеопроцессоров с полным набором телевизионных функций TDA935x/6x/8x.

Телевизионные процессоры системы управления



2-е издание,
ISBN: 5-94387-047-4
Формат: 165×235 мм
Объем: 512 с.
Цена: 83 руб.

Заказ____экз.

Журавлев В.А.

В книге собраны материалы, касающиеся более ста наиболее распространенных семейств ТВ. В содержании книги в алфавитном порядке представлены: тип управляющего микроконтроллера, фирмы-производители, типы ТВ, в которых он применен. Такая концепция имеет своей целью облегчить поиск необходимого процессора и способа его программирования с ПДУ.

Принимаются ксерокопии.

Закажите лучшие книги для телемастеров

1001 секрет телемастера
в трех книгах



2-е издание,
перераб. и доп.

ISBN: 5-94387-170-5
Формат: 165×235 мм
Объем: 304 с.
Цена: 153 руб.

Заказ____экз.

2-е издание,
перераб. и доп.

ISBN: 5-94387-186-1
Формат: 165×235 мм
Объем: 224 с.
Цена: 153 руб.

Заказ____экз.

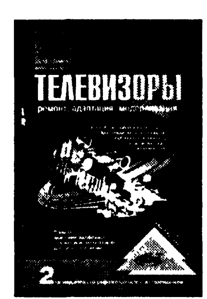
ISBN: 5-94387-196-9
Формат: 165×235 мм
Объем: 256 с.
Цена: 153 руб.

Заказ____экз.

Рязанов М.Г.

Созданию трехтомника предшествовал многотысячный поток электронных писем со всего мира на сайт автора www.telemaster.ru с просьбой дать совет по ремонту или с рассказом о том, как были решены проблемы с ремонтом зарубежных и отечественных ТВ. Работает форум телемастеров. Секреты ремонта в книгах систематизированы в алфавитном порядке. В книгах даны также фрагменты схем, описан состав шасси.

Телевизоры: ремонт, адаптация, модернизация



Саулов А.Ю.

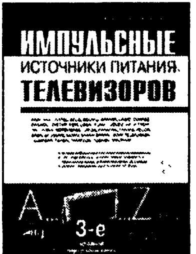
Эта книга рассчитана на радиолюбителей с небольшим стажем, которые хотят испытать свои силы в модернизации и ремонте своего любимого телевизора. Описаны методы ремонта отечественных и зарубежных телевизоров, адаптация «европейских» телевизоров под наши стандарты.

2-е издание,
перераб. и доп.

ISBN: 5-94387-171-3
Формат: 140×205 мм
Объем: 336 с.
Цена: 125 руб.

Заказ____экз.

Импульсные источники питания телевизоров



Рязанов М.Г.
Янковский С.М.

В книге кратко рассмотрены принципы построения и работа источников питания ТВ.

Основной упор сделан на рассмотрение конкретных схем с приведением необходимых пояснительных материалов. Схемы источников питания систематизированы по применяемой для их построения элементной базе.

Специальный раздел посвящен секретам ремонта, где рассотрено несколько сотен реальных неиспраностей.

3-е издание,
перераб. и доп.

ISBN: 5-94387-353-8
Формат: 165×235 мм
Объем: 400 с.
Цена: *** в печати

Заказ____экз.

Принимаются ксерокопии.

(принимаются ксерокопии)

Автор	Название	Цена Россия (руб.)	Цена Украина (грн.)	Год	Объем	Заказ (экз.)
Колисниченко	Новичок за рулем. Советы психолога, юриста, инструктора	87	19	2006	368	
Левенбаум	Надо ли худеть? Как стать красивой. Рецепты, советы, рекомендации	65	15	2006	320	
Цветкова	Кладовая здоровья на вашем столе: фрукты	47	11	2006	240	
Цветкова	Кладовая здоровья на вашем столе: овощи	43	10	2006	208	

Серия "Кратко о важном"

Автор	Название	Цена Россия (руб.)	Цена Украина (грн.)	Год	Объем	Заказ (экз.)
Башкирова	Ждем ребенка. Рекомендации, подсказки, советы	22	5	2006	128	
Бердникова	Здоровый малыш (100 практических советов по уходу за ребенком)	22	5	2006	128	
Башкирова	Ваш малыш - это личность (100 практических советов по воспитанию ребенка)	22	5	2006	128	
Левенбаум	Самые популярные Диеты! За и против	22	5	2006	128	
Левенбаум	Диеты! Худеем по-вегетариански	22	5	2006	128	

Компьютерная литература

Серия: Компьютерная шпаргалка

Автор	Название	Цена Россия (руб.)	Цена Украина (грн.)	Год	Объем	Заказ (экз.)
Егоров	МиниЖелтые страницы Интернет. Компьютерная шпаргалка	18	5	2006	80	
Егоров	Поиск в Интернет. Компьютерная шпаргалка	18	5	2006	80	
Золотарева	Электронная почта. Компьютерная шпаргалка	18	5	2006	80	
Колосков	Microsoft Windows XP. Компьютерная шпаргалка	18	5	2006	80	
Кузнецова	Microsoft Word 2003: работаем с текстом	18	5	2006	80	
Матвеев	Вычисления и расчеты в Excel 2003. Комп. шпаргалка	18	5	2006	80	
Юдин	Microsoft Excel 2003: работаем с таблицами	18	5	2006	80	

Серия: Просто о сложном

Автор	Название	Цена Россия (руб.)	Цена Украина (грн.)	Год	Объем	Заказ (экз.)
Алешков	Программы-переводчики. Осваиваем сами	54	12	2005	144	
Антоненко	Тонкий самоучитель работы на компьютере + цветные вклейки	87	19	2007	256	
Антоненко	"Толстый" самоучитель работы на компьютере, 2-е изд.	149	33	2007	544	
Бруга	Java по-быстрому. Практический экспресс-курс	164	36	2006	384	
Вольский	Turbo Pascal 7.0 для студентов и школьников	83	18	2007	224	
Воробьев	Nero Burning ROM 7. Записываем CD и DVD	65	15	2007	192	
Дмитриев	Настройки BIOS, 3-е изд., перераб. и доп.	87	19	2007	288	
Егоров	Легкий самоучитель работы в Интернете. Все самое необходимое + цветные вклейки	87	19	2006	256	
Жарков	AutoCAD 2007. Эффективный самоучитель	219	***	2007	608	
Жарков	AutoCAD 2004. Эффективный самоучитель. Изд. 2-е	164	37	2005	560	
Жарков	AutoCAD 2005: Эффективный самоучитель	173	39	2005	600	
Жарков	AutoCAD 2006: официальная русская версия. Эффективный самоучитель	186	42	2006	592	
Жарков	Создаем чертежи в AutoCAD 2006/2007 быстро и легко	98	22	2007	256	
Золотарева	Желтые страницы Интернет 2006: Лучшие русские ресурсы	94	20	2006	368	
Кадлец	DELPHI: Книга рецептов. Практические примеры, трюки, секреты	164	36	2006	384	
Кальвик Дэвид	3Ds Max 8: осваиваем на практике создание трехмерных миров + цветные вклейки	197	44	2006	368	
Колисниченко	Самоучитель PHP 5. 3-е издание	182	40	2005	576	
Колисниченко	Самоучитель LINUX. Установка, настр., использ. Изд. 4-е.	175	39	2006	688	
Колисниченко	Сделай сам комп. сеть. Монтаж, настройка, обслуж. Изд. 2-е	142	29	2004	448	
Колисниченко	Англо-русский толковый словарь компьютерных терминов	83	18	2006	272	
Куприянова	Ядерные кнопки. Приемы эффективной работы с использованием горячих клавиш	43	10	2007	128	

(принимаются ксерокопии)

Автор	Название	Цена Россия (руб.)	Цена Украина (грн.)	Год	Объем	Заказ (экз.)
Колосков	Windows XP. Популярный самоучитель. Изд. 2-е. перер. и доп.	108	24	2005	368	
Кузьмин	Поиск в Интернете: Как искать, чтобы найти	54	12	2006	160	
Кузнецова	Установка и переустановка Windows. Изд. 5-е.	***	***	2007	128	
Кузнецова	Microsoft Windows XP. Краткое руководство	63	14	2005	256	
Куприянова	Реестр Windows XP: Трюки, настройки, секреты	65	15	2006	192	
Лохниски	222 проблемы работы на компьютере и их решение	65	39	2006	224	
Марек	Ассемблер на примерах. Базовый курс	109	24	2005	240	
Матвеев	Самоучитель MS Windows XP. Все об использ. и настр. Изд. 2-е	160	36	2006	624	
Моркес	Microsoft Access 2003. Эффективный самоучитель	164	37	2006	352	
Подольский	Печать на ПК слепым десятипальцевым методом. Изд. 3-е	28	6	2006	96	
Пономарев	Самоучитель работы на ПК + цветные вклейки, 2-е изд.	109	24	2007	368	
Серогодский	Excel 2003 + **цв.вклейки**. Эффективный самоучитель. Изд. 2-е.	153	34	2006	400	
Сухарев	Turbo Pascal 7.0. Теория и практика программирования, 3-е изд.	164	37	2007	544	
Юдин	Легкий самоучитель работы на ноутбуке + цветные вклейки	109	24	2007	256	
Юдин	Самоучитель работы на ноутбуке, 3 -е изд. перераб. и доп.	186	42	2007	512	
Юдин	Самоучитель работы на ноутбуке. Изд. 2-е + **цв.вклейки**	175	39	2006	512	
	Скачиваем фильмы, музыку и программы из Интернета. Пиринговые сети	65	15	2006	272	

Серия: Полное руководство

Автор	Название	Цена Россия (руб.)	Цена Украина (грн.)	Год	Объем	Заказ (экз.)
Бен Лонг	Цифровая фотография от А до Я. Полное руководство с цв. вклейками **+CD**	285	65	2006	592	
Досталек	TCP/IP и DNS в теории и на практике. Полное руководство	263	58	2006	608	
Колисниченко	Linux: Полное руководство	252	54	2006	784	
Шетка	Microsoft Windows Server 2003: Полное руководство	241	53	2006	608	

Серия: Секреты мастерства

Автор	Название	Цена Россия (руб.)	Цена Украина (грн.)	Год	Объем	Заказ (экз.)
Колисниченко	IRC, IRC-каналы, IRC-боты: как пользоваться и как сделать самому . Избранные технологии Интернета	186	42	2006	368	
Колисниченко	Linux-сервер своими руками. Изд. 4-е перер. и доп.	219	45	2005	752	
Колисниченко	Rootkits под Windows. Теория и практика программирования "шапок-невидимок", позволяющих скрывать от системы данные, процессы, сетевые соединения.	175	39	2006	320	
Мозговой	Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход	175	37	2006	320	
Мозговой	С++Мастер-класс. 85 нетривиальных проектов, решений и задач	193	43	2007	272	
Смит	Оптимизация и защита Linux-сервера своими руками	219	49	2006	576	
Сухарев	Основы Delphi. Профессиональный подход	184	37	2004	600	
Финков	Интернет. Шаг второй: от пользоват. к профессионалу + **CD**	88	17	2002	768	
Юдин	Ноутбук: особенности использования и настройки. + **цв.вклейки.** 2-е изд., перераб. и доп.	186	41	2006	416	

Серии: Профи и др.

Автор	Название	Цена Россия (руб.)	Цена Украина (грн.)	Год	Объем	Заказ (экз.)
Вебер	Knowledge-технологии в консалтинге и управл.предпр.+ **CD**	127	18	2003	176	
Гургенидзе	Мультисервисные сети и услуги широкополосного доступа	87	18	2003	400	
Есипов	Информатика (учебник). Изд. 3-е	102	21	2003	400	
Куприянов	Техническое обеспечение цифровой обработки сигналов	66	12	2000	752	
Кучеров	Источники питания ПК и периферии. Изд. 3-е	142	32	2005	432	
Кучерявый	Пакетная сеть связи общего пользования	109	27	2004	272	
Кучерявый	Управл. трафиком и качество обслуживания сети Интернет	128	29	2004	336	
Щеглов	Защита комп. информации от несанкционир. доступа	164	20	2004	384	

(принимаются ксерокопии)

Автор	Название	Цена Россия (руб.)	Цена Украина (грн.)	Год	Объем	Заказ (экз.)
Серии: Конспект программиста и Библиотека пользователя						
Будилов	Практические занятия по PHP 4 + **CD**	33	9	2001	352	
Будилов	Работаем с FINALE 2001 + **CD**	50	10	2001	240	
Костельцев	Построение интерпретаторов и компиляторов + **CD**	44	9	2001	224	
Куправа	Самоучитель Access 97/2000 + **дск**	43	8	2001	144	
Николенко	MIDI — язык богов + **CD**	39	8	2000	144	
Радиоэлектроника, электричество и связь						
Серия: Домашний мастер						
Андреев	Аудиосистема класса Hi-Fi своими руками: советы и секреты	131	29	2006	208	
Балясников	Обувь. Выбор, уход, ремонт	32	9	2003	240	
Бадьин	Справочник Строителя-технолога	132	30	2005	320	
Бирюков	Защита автомобиля от угона	61	10	2003	176	
Давиденко	Настольная книга дом. электрика. Люмин. лампы.	109	18	2005	224	
Дворецкий	Автомобильные сигнализации (модели E...Z)	175	29	2006	544	
Корякин-Черняк	Холодильники от А до Я. Изд. 2-е, перераб. и дополн.	123	20	2003	416	
Корякин-Черняк	Освещение квартиры и дома	87	19	2005	192	
Корякин-Черняк	Краткий справочник домашнего электрика. Изд. 2-е	98	22	2006	272	
Корякин-Черняк	Современные автосигнализации (модели А...Е)	158	29	2006	400	
Корякин-Черняк	Справочник домашнего электрика 5-е изд. перераб. и доп.	***	***	2007	400	
Ландик	Современные холодильники NORD	72	10	2003	144	
Пестриков	Новейшая азбука сотового телефона. Изд. 3-е	142	22	2005	352	
Пестриков	Домашний электрик и не только. Кн. 1. Изд. 5-е	98	22	2006	224	
Пестриков	Домашний электрик и не только. Кн. 2. Изд. 5-е	98	22	2006	224	
Торопкин	Ламповый Hi-Fi усилитель своими руками. Изд. 2-е	131	27	2006	272	
Серия: Радиолюбитель						
Белов	Конструирование устройств на микроконтроллерах	нет	20	2005	256	
Виноградов	Микропроцессорное управление телевизорами	55	11	2003	144	
Петров	Звуковая схемотехника для радиолюбителей	нет	27	2003	400	
Саулов	Телевизоры: ремонт, адаптация, модернизация. Изд. 2-е	125	28	2005	320	
Саулов	Металлоискатели для любителей и профессионалов	109	24	2004	224	
Семьян	500 схем для радиолюбит. Приемники. Изд. 2-е перер. и доп.	109	24	2005	272	
Семьян	500 схем для радиолюбителей. Источники питания. Изд. 2-е	142	25	2006	416	
Семьян	500 схем для радиолюбителей. Радиостанции и трансиверы	120	27	2006	272	
Шмырев	Радиостанция своими руками + **вклейка**	87	19	2004	144	
Серии: Телемастер и Энциклопедия телемастера						
Безверхний	Телевизоры DAEWOO и SAMSUNG + **схемы**	153	17	2003	144	
Виноградов	Импульсные источники питания видеомагнитофонов	65	9	2003	160	
Корякин-Черняк	Применение телевизионных микросхем. Т. 1	164	22	2004	320	
Корякин-Черняк	Применение телевизионных микросхем. Т. 2	164	22	2004	304	
Корякин-Черняк	Применение телевизионных микросхем. Т. 3	164	22	2005	320	
Пьянов	Тел. LG на шасси MC-51B, MC-74A, MC-991A + **схемы**	131	13	2003	144	
Пьянов	Видеопроцессоры семейства UOC + **схемы**	109	13	2003	160	
Рязанов	1001 секрет телемастера. Кн. 2. Изд. 2-е, перер. и доп.	164	34	2005	224	
Рязанов	1001 секрет телемастера. Книга 3	164	34	2006	256	
Рязанов М.Г.	1001 секрет телемастера. Книга 1, 3-е изд., доп. и перераб.	164	37	2007	288	
Рязанов М. Г.	Импульсные источники питания телевизоров, 3-е изд. перераб. и доп.	175	39	2006	400	
Саулов	Переносные телевизоры	142	12	2002	512	

Вышли новые книги «500 схем для радиолюбителей»

500 схем для радиолюбителей. **Шпионские штучки и не только...**



Полезная подборка схемных решений. Радиомикрофоны. Устройства для снятия информации с оконного стекла. Устройства записи телефонных переговоров и защиты от записи. Обнаружители жучков и диктофонов. Постановщики помех. Автобезопасность. Измерительные приборы и многое другое.

500 схем для радиолюбителей. **Электронные датчики**



В данной книге представлены схемные решения электронных датчиков, т.е. описаны конструкции устройств, позволяющих организовать охрану комнаты и автомобиля, защитить помещение от пожара, выявить наличие жучков. Приводимого краткого описания вполне достаточно для самостоятельного изготовления понравившейся конструкции.

500 схем для радиолюбителей. **Дистанционное управление моделями**



Данная книга уникальна. Она познакомит читателя с принципами функционирования и практической схемотехникой. Все рассмотренные конструкции выполнены на современной элементной базе, схемы сопровождаются подробными описаниями, рисунками печатных плат, рекомендациями по сборке и настройке.

Россия: С.-Петербург, пр. Обуховской обороны, 107,
для писем: 192029 Санкт-Петербург а/я 44
(812)-567-70-25, 567-70-26, E-mail: nit@mail.wplus.net

Украина: 02166, Киев-166, ул. Курчатова, 9/21,
(044)-516-38-66, E-mail: nits@voliacable.com



www. nit.com.ru